National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

NL-339 (r. 88/04) c

Canada

UNIVERSITY OF ALBERTA

Communication in FLEX

by

© *Yan Li*

A thesis

submitted to the Faculty of Graduate Studies and Research

in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Edmonton, Alberta

Spring, 1990

National Library of Canada
Bibliothèque nationale du Canada

Canadian Theses Service
Service des thèses canadiennes

Ottawa, Canada
K1A 0N4

NOTICE

The quality of this microform is heavily dependent upon the quality of the original thesis submitted for microfilming. Every effort has been made to ensure the highest quality of reproduction possible.

If pages are missing, contact the university which granted the degree.

Some pages may have indistinct print especially if the original pages were typed with a poor typewriter ribbon or if the university sent us an inferior photocopy.

Reproduction in full or in part of this microform is governed by the Canadian Copyright Act, R.S.C. 1970, c. C-30, and subsequent amendments.

AVIS

La qualité de cette microforme dépend grandement de la qualité de la thèse soumise au microfilmage. Nous avons tout fait pour assurer une qualité supérieure de reproduction.

S'il manque des pages, veuillez communiquer avec l'université qui a conféré le grade.

La qualité d'impression de certaines pages peut laisser à désirer, surtout si les pages originales ont été dactylographiées à l'aide d'un ruban usé ou si l'université nous a fait parvenir une photocopie de qualité inférieure.

La reproduction, même partielle, de cette microforme est soumise à la Loi canadienne sur le droit d'auteur, SRC 1970, c. C-30, et ses amendements subséquents.

ISBN 0-315-60276-7

NL-339 (r. 88/04) c

Canada

THE UNIVERSITY OF ALBERTA

RELEASE FORM

NAME OF AUTHOR: *Yan Li*

TITLE OF THESIS: *Communication in FLEX*

DEGREE: *Master of Science*

YEAR THIS DEGREE GRANTED: *1990*

Permission is hereby granted to The University of Alberta Library to reproduce single copies of this thesis and to lend or sell such copies for private, scholarly or scientific research purposes only.

The author reserves other publication rights, and neither the thesis nor extensive extracts from it may be printed or otherwise reproduced without the author's written permission.

(Signed) .....Yan Li.....

Permanent Address:
11251-35 Ave.
Edmonton, Alberta
Canada T6J 3M8

Date: December 28, 1989

UNIVERSITY OF ALBERTA

FACULTY OF GRADUATE STUDIES AND RESEARCH

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies and Research, for acceptance, a thesis entitled **Communication in FLEX** submitted by **Yan Li** in partial fulfillment of the requirements for the degree of **Master of Science**.

Supervisor: M. Tamer Ozsu

Committee member: Ahmed Kamal

Committee member: Jack Mowchenko

Committee member: Li-Yan Yuan

Date: Dec. 26, 89

To my parents

ABSTRACT

The project (**FLEX**) has as its aim the creation of an object-oriented operating system which will be used in research on the architectural as well as algorithmic problems associated with integrating distributed operating systems and distributed database management systems. The focus of this thesis is on one component of this operating system: computer communication in an object-oriented environment.

The thesis presents the design of a communication facility for **FLEX** and its prototype implementation. It also contains a second variant of the design of **FLEX**, which emerged as the implementation details were being worked out.

FLEX is implemented as a prototype system running as a collection of user processes under the control of a time-sharing operating system (*UNIX*). The communication facility is based on the LLC layer, as described in standard 802.2. The lower layers are implemented using *UNIX* primitives.

ACKNOWLEDGEMENTS

It has been a heavy-weight process (see p.88).

I must thank Jasmine and the rest of my family for their cooperation and endurance.

It would have been impossible for me to finish the thesis today without the support from Mr. and Mrs. OuYang.

I would also like to thank my supervisor, Tamer Ozsu, for his patience, the committee members for their helpful comments, and Keith Fenske for his help along the way.

TABLE OF CONTENTS

1 Introduction 1

1.1 Motivation 1

1.2 Distributed DBMS Requirements 2

1.3 Scope and Organization of the Thesis 6

2 Background 8

2.1 Communication in a Distributed Environment 8

2.2 Communication Schemes 9

2.3 Techniques in Implementing Communication in an Operating System 15

2.4 Networks 18

2.5 Review of Distributed Operating Systems 28

3 Overview of FLEX 41

3.1 Design Philosophy of FLEX 41

3.2 FLEX Paradigm 43

3.3 Kernel Design 46

3.4 Non-Kernel Design 54

4 Design of the Communication Module in FLEX 56

4.1 Communication Scheme Adopted in FLEX 56

4.2 Design Issues 57

4.3 Network Interface in FLEX 60

4.4 Classification of Communication Partnerships 61

4.5 Semantics of Communication 64

5 Implementation of the Communication Module in FLEX 67

5.1 Implementation Details 67

5.2 Comments on Implementation 77

6 Recommendations for Improvement 79

6.1 Major Changes to the Previous Design 79

6.2 The Proposed System Design 84

6.3 Overview of the Communication Design 105

7 Conclusions 108

7.1 Notes on the Prototype Implementation 108

7.2 Experience Gained from the Research 109

7.3 Recommendations for Future Work on the Project 111

Bibliography 112

LIST OF FIGURES

Figure 2.1: The Comparison of Message and Procedural Approach 18

Figure 2.2: Frequency-division Multiplexing with Time-division Multiplexing 20

Figure 2.3: The Network Architectures 24

Figure 2.4: 802 Protocol Layers Compared to OSI 26

Figure 2.5: Remote Communication in Mach 35

Figure 2.6: An Example of a File Access in LOCUS 39

Figure 2.7: The Newcastle Connection 40

Figure 3.1: FLEX Architecture 44

Figure 3.2: An Example of an Object Structure 48

Figure 3.3: Object Hierarchy 49

Figure 3.4: An Active Object 50

Figure 3.5: An Example of Object Sharing with Different Capabilities 52

Figure 4.1: Communication Partnerships in FLEX 62

Figure 5.1: Structure of an Object Descriptor 69

Figure 5.2: Format of a Message 72

Figure 5.3: Structure of a Supermailbox 74

Figure 6.1: FLEX Kernel 86

Figure 6.2: FLEX System Architecture 96

CHAPTER 1

Introduction

This thesis is part of an ongoing research on the design and implementation of a distributed database operating system. This project, named **FLEX**, has two main aims:

(a) to investigate the architecture of a distributed operating system which can accommodate both general computing requirements and the fundamental database constructs, and

(b) to look into the design and analysis of various operating system functions in order to make them more suitable for database operations.

My contribution to the **FLEX** project is three-fold:

(1) participation in the overall design of **FLEX** ([OzLLT-88]),

(2) design and implementation of the communication facility of **FLEX**, and

(3) a new, more general, design of the **FLEX** system.

This work is reflected in this thesis: chapter 3 contains a presentation of the initial design, as described in [OzLLT-88], chapters 4 and 5 contain a presentation of my communication facility and its implementation, and, finally, chapter 6 describes my new design proposal.

1.1. Motivation

Distributed systems are very fashionable nowadays; this trend is not unreasonable. There are numerous arguments supporting the use of distributed systems in many circumstances.

Among the more commonly mentioned contexts in which distributed systems are applicable are database systems.

Traditional database systems are built on top of existing operating systems that were not designed with the requirements of database systems in mind. The lack of a reasonable paradigm for co-operation between operating systems and database management systems (DBMS) results in duplication of some common services leading to a degradation in system performance. For example, the services required by a DBMS, such as synchronization, data and file management, may be provided by an operating system but at a different level. Thus, some of the functions have to be re-implemented within the DBMS. This is particularly true of distributed operating systems; there still remains insufficient concern for providing the necessary support for distributed database applications.

In a distributed setting, new problems are introduced on top of the problems already existing in centralized DBMSs. The issues like transparency, naming, fault tolerance, etc., and especially communication and concurrency control have to be dealt with.

More specific comments on the motivation for constructing operating systems for database applications are presented in [Ozsu-88]. These concerns are summarized below.

1.2. Distributed DBMS Requirements

1.2.1. Cooperation with the Distributed Operating System and Networks

The coupling of distributed database managers and distributed operating systems is not simple. All the traditional problems of interfacing the DBMS with its operating system remain; more over, communication network protocols also need to be considered. The standards that are being developed for communication networks and those used in the design of

distributed systems are incompatible. Thus, porting a DBMS to a different networking environment can result in major re-work to accommodate the idiosyncrasies of each individual network architecture. While interfacing the operating system with a network standard at a relatively high level is essential for portability and compatibility with developments in networking standards, the architectural paradigm should accommodate the basic axioms of distributed DBMS functions.

1.2.2. Some DBMS Axioms

(1) Efficient management of the buffer pool.

The speed of accessing data buffers is a critical factor in the overall efficiency of the DBMS, whether distributed or not. Alas, the buffer management algorithms used inside a DBMS often collide with the algorithms used by memory management of the underlying operating system. Hence this is a much bigger issue than it would seem at first.

(2) Access transparency.

Distributed DBMSs typically require that access to the database(s) be transparent, even though the database itself is distributed. A distributed DBMS should provide not only data independency (i.e., changing data does not result in a change of application programs), but also distribution and replication transparencies.

(3) Access control.

The distributed DBMS must ensure that any access to the database is subject to access control, for the purpose of protecting data and information from unauthorized persons.

(4) Reliable remote communication.

A distributed DBMS must use a reliable remote communication mechanism. This

mechanism should not compromise transparent access to remote resources, as mentioned earlier.

(5) Compatibility with networking protocols.

This permits the delegation of a considerable amount of the communication work to network software. It should be pointed out that this is a controversial requirement not necessarily accepted by all the researchers.

(6) Transaction management.

The operation of a transaction manager requires good concurrency control and a recovery mechanism.

1.2.3. Support from the Distributed Operating System

The previous section listed some axioms that apply to the operation of a DBMS. In principle, these axioms could be satisfied by the DBMS directly, perhaps with a small loss of efficiency. From a practical point of view, however, it is preferable to look for support for a large part of them to the software that operates underneath the DBMS, i.e., the operating system. This system support is needed in several areas:

(1) Memory management.

The memory management policy of the operating system may support the operation of the buffer pool manager; on the other hand, it may also reduce its efficiency, if the two are not well co-related. Thus, the buffer pool manager must be compatible with the algorithms of the virtual memory manager (in the operating system); likewise, the operating system must leave enough flexibility to make the existence of a separate buffer pool manager worthwhile.

(2) File system.

Besides problems related directly to buffering, file system properties are at the root of a few other, perhaps less spectacular problems. Among them are: locking and its granularity, advanced access methods, etc. A primary example of a system that fails to support DBMS applications on all these counts is *UNIX*.

(3) Naming.

Naming is needed for maintaining transparent access to data and for replication support; thus, a distributed DBMS would expect the distributed operating system to implement a naming mechanism that provides transparent access to logical as well as physical system resources.

As a convenient byproduct, a good naming mechanism makes data sharing possible.

(4) Access control and protection.

The DBMS should use the access control mechanism provided by the operating system, provided that it is sufficiently general. Ideally, the operating system should offer a capability-based access control method. The same should apply to protection.

(5) Communication.

Efficient and powerful communication primitives implemented in a reliable fashion are welcome. In particular:

(a) do not lose messages, and

(b) return all undeliverable messages to the sender.

These requirements point to the necessity of proper message buffering and hand-shaking to ensure reliable host-to-host delivery.

(6) Concurrency control.

Concurrency control is provided by every operating system, but not necessarily in a way that satisfies DBMS applications. Traditionally, the DBMS builds its own concurrency control on top of that provided by the operating system. This causes an unnecessary redundancy and inefficiency. Recent developments in operating systems seem to go in the right direction: concepts such as heavyweight/lightweight processes and threads are sufficiently flexible to be used by the DBMS directly.

(7) Transaction management support.

In current DBMSs, the transaction manager is implemented as part of the DBMS. However, making transaction management part of the standard operating system services would permit using the concept of a transaction not only by the distributed DBMS, but also any application that runs on the operating system. Such a support would require that concurrency control and recovery primitives be implemented within the operating system.

1.3. Scope and Organization of the Thesis

The emphasis of this thesis is on the communication aspect of the system. A suitable communication module is designed and implemented in the context of the existing **FLEX** design. As a result of experimenting with the **FLEX** design during this research, a new version of the design of **FLEX** is proposed.

The thesis is organized as follows:

Chapter 2 gives a general discussion on communication in operating systems, including a brief review of some existing systems.

Chapter 3 describes the version of the **FLEX** design that the implementation of communication is based on. This design is also presented in both [OzLLT-88] and [Lau-88].

Chapters 4 and **5** present the design and the implementation of communication in **FLEX**.

Chapter 6 proposes a new version of the design of **FLEX**; it includes highlights of several major changes and a formal description of the system architecture.

Chapter 7 summarizes the thesis and provides a discussion of the ongoing research.

CHAPTER 2

Background

Many distributed operating systems have been developed for experimenting with distributed computing. Since the major issue addressed in this thesis is communication, this chapter provides some basic background information on the way that such systems address local and network communication, namely on:

- Communication schemes,
- Implementation of communication schemes in operating systems,
- Network architectures, and
- Communication standards.

The chapter concludes with a presentation of the highlights of a few features of some existing systems that are related to the design of **FLEX**.

2.1. Communication in a Distributed Environment

A computer network is "an interconnected collection of autonomous computers"[Tanen-81]. Communication in a distributed environment can be either local or remote (or network) communication.

Local communication occurs between entities (e.g., processes, device drivers, etc.) on the same machine. It involves only one operating system kernel. On the other hand, network communication happens when an entity on one machine wants to exchange information with an entity on another machine or accesses (physical or logical) resources of another machine.

Such communication involves the network connecting the machines and more than one operating system kernel. To provide a good understanding of the issues involved in communication, the following sections (2.2, 2.3, and 2.4), will first present various communication schemes, then discuss some of the techniques used in implementing communication facilities, and finally look at network communication standards.

2.2. Communication Schemes

In typical operating systems, the basic active entity in the system is the **process** (including daemons[1]). Communication between processes takes on three basic forms:

- Message-based, or **passive** communication.
- Interrupt-driven, or **active** communication,
- Network communication, which is an unreliable version of passive communication.

Message-based communication permits the cooperation of independent user entities. A receiver responds to a message only if it wants to. Network communication is a weak variation of passive communication: not only the receiver is not obliged to respond to the request, but there is no obligation on the part of the kernel (on the receiving side) to forward the request to the receiver.

Interrupt-driven communication is used when the communication is in the form of imperative requests. Context switching occurs between user entities and/or the kernel as a result of such a request.

---

[1]Daemons are server processes that are not associated with any users but do system-wide functions[Bach-86].

Active and passive communication schemes exist in almost every general-purpose operating system. In some systems, passive and active communication schemes are somewhat artificially unified: interrupt-driven communication is presented as a form of message passing. In such systems, a context switch can be forced by sending a message to the scheduler and entering a wait state (e.g., the so-called "blocking send").

Besides the above, systems must allow some form of communication between processes and kernel modules. Underneath all of it is yet another communication level: communication inside the kernel[2] (i.e., between kernel modules). This lowest level will not be discussed here, as it is transparent to higher levels, and, as such, is only a matter of implementation expedience.

When a sender addresses a receiver by **naming** it, it communicates with the receiver **directly**. As this is not always convenient or feasible (e.g., the name of the receiver is unknown or the receiver has not yet been created), there is another form of communication that does not require naming the receiver: **indirect communication**.

Note that in some systems, specifically those that are capability-based, a sender does not use the name of the receiver when addressing it; instead, it refers to the receiver by showing a special indirect access **handle**, called **capability**. Although such a capability has the form of an encoded string, it really does represent the receiver in a unique way, i.e., given a capability, the kernel can determine which receiver is in question. Thus, a capability is just an indirect way of naming a receiver, with an additional level of protection.

---

[2]A kernel can be viewed as one large, monolithic, event-driven program[BlMaM-87].

2.2.1. Passive Communication

Passive communication, denoted by

$$"P \rightarrow Q"$$

takes place when entity P sends a message to entity Q, **requesting** some service. The message is placed in a message queue, usually following a first-in-first-out discipline. Q is not forced to perform the service; it is not even required to accept the request at all. Thus, the fact that a sender sends a message does not imply that the receiver will ever accept the message or, in other words, does not imply that the message will ever be delivered.

There are many advantages of passive communication. The most important are:

- It permits anonymous processes to communicate, which is an easy way of resolving the difficult naming problems. This is done by using mailbox-like objects, which are intermediate repositories for messages.
- As it is a form of lazy binding (on demand by the receiver), it removes simple timing problems that plague the active communication scheme. The producer-consumer relationship is a prime example of this situation. If the consumer is created after the producer sends the first message (due to scheduling), passive communication will work correctly, while active communication will fail, having no one to interrupt.
- As messages are received on demand, the receiving process can organize its flow of control. This is particularly important in cases when a process has more than one communication partner (e.g., a server process).
- No messages are ever lost in passive communication. They may remain undelivered, but only because of a lack of interest on the part of the receiver.

Passive communication is not free from deficiencies. Messages have to be queued, it is impossible to guarantee the delivery of a message, and blocking receive may result in unnecessary waiting. These deficiencies make the passive communication a clumsy synchronization tool, especially in contrast to active communication.

Although other queuing disciplines could be considered, only one is used in practice: FIFO. It will be assumed here that FIFO queuing is used in passive communication.

2.2.2. Active Communication

Active communication between two entities P and Q, denoted by

$$"P \Rightarrow Q"$$

takes place when entity P **forces** entity Q to perform some action by issuing a **service call** (e.g., *system call* or *procedure call*).

Active communication immediately alters the flow of control: a context switch is performed at the very instant of communication, passing control from the sender to the receiver (i.e., from P to Q). This control switch does not resume Q at the point where it was deprived of the CPU. Instead, it creates a new and independent operation through the code of Q. In particular, if Q is already in the midst of responding to an earlier demand (i.e., an earlier instance of active communication), but lost the CPU for one reason or another, the new demand will be executed next. This phenomenon is known as **stacking** (as opposed to **queuing** in passive communication).

The main advantage of using active communication is the guarantee of delivery. This simplifies greatly the code of the sender process, which may treat the operation of delivering a message as a single indivisible operation. As an indirect consequence of this fact, there is no

need for message buffers; moreover, active communication serves as an excellent synchronization tool, as delivery of a message is immediate and guaranteed.

Systems that use active communication have to resolve a number of issues that are not relevant to passive communication. One is naming: the sender must address the receiver directly by name. Another is message stacking: a new message may arrive when its predecessor is being processed. Naive code would almost certainly fail in such a situation (or at least introduce an inconsistency); code handling message stacking correctly has to be quite complex.

2.2.3. Network Communication

Communication to a remote site is performed by sending messages through a network link. The very nature of the hardware makes active communication impractical, as the very existence of the receiver can never be guaranteed. Thus, network communication is very similar to passive communication and could be regarded as such.

It is, however, convenient to consider network communication as a special communication scheme. In passive communication, the sender can rely on checking being done by the kernel: the sender knows that if a message is not rejected by the kernel immediately, it is available to the receiver. Likewise, a potential receiver informed that there are no pending messages can be sure that no one has sent any. This is certainly not true of network communication.

The main property of network communication is that messages may be lost. The kernels involved (two or more, as messages go through a network) can not determine in a direct way whether a message was lost or not. They can only do it from contextual information, which in

turn, can not be proven correct.

Note that the above remarks relate to the operation of sending individual messages. In many networks, there are additional network layers responsible for making network communication statistically reliable. This is achieved by acknowledgments, retransmission of messages, etc. But even such precautions can not completely guarantee that messages are never lost, unless one assumes that the networking hardware and software are both immune to faults[3].

2.2.4. Communication with Kernel Modules

In principle, either passive or active communication could be used in communicating with kernel modules. In reality, all the systems that have a concern for efficiency adopt active communication to avoid polling or busy wait. When a process wants to ask the kernel for a service, it triggers a software interrupt (*svc*), which forces the kernel to respond immediately.

2.2.5. Direct and Indirect Communication

Entities may communicate with each other **directly** by exhibiting the appropriate capabilities. Such a communication scheme requires that all the partners involved are known to each other and labeled directly. Sometimes, objects prefer **indirect** communication, because it enables anonymous cooperating objects to exchange information without having to specify their partners' identities; this is accomplished *via* **mailboxes**.

Therefore, one may envision three different communication schemes:

(1) Direct communication: all the partners know precisely whom they want to communicate with and identify their partners directly, using a capability, which is internally converted

[3]This is known as the *Generals' problem*[Gray-78].

into the name of the partner.

(2) Consistent indirect communication: all the partners remain anonymous throughout the whole communication session. Their only meeting ground is in the form of a named mailbox (the name of this mailbox is needed to distinguish among the many unrelated communication cooperatives).

(3) A mixed scheme, in which objects start as anonymous entities, but may choose to switch to direct communication with a subset of the partners involved in the communication cooperative.

2.3. Techniques in Implementing Communication in an Operating System

Interprocess communication (IPC) is an essential part of an operating system. It allows arbitrary processes to exchange data and synchronize execution.

For processes on the same machine, IPC takes many forms, e.g.,

(1) **message passing** to allow processes to send formatted data streams to arbitrary processes,

(2) **pipes** to permit a reliable uni-directional data stream between processes,

(3) **semaphores** to allow processes to synchronize execution,

(4) **shared files** to allow processes to share a common data area,

(5) **shared memory** to allow processes to share parts of their virtual address space, and

(6) **signals** to provide software interrupts to given processes.

Unfortunately some of the methods used for local process communication are not suitable in a distributed environment. As a result, IPC becomes one of the major issues in

distributed operating systems design. Many systems have been developed to investigate and experiment with different IPC facilities as discussed in great detail in [TanvR-85]; here we will look into a few of the concepts involved.

(1) Message Passing vs Shared Memory

In a centralized environment, a message passing system allows cooperating processes to exchange messages. The operating system is responsible for communication, that is to move a message from the sender's address space to the receiver's address space, possibly involving the kernel's address space for buffering.

Shared memory systems allow communicating processes to share a block of variables; thus, application programmers shoulder more responsibility for organizing communication. Another possibility is to use the registers of a processor for fast interprocess communication[Cheri-84_2]. If a message is to be transferred from process A to process B, the processor's general registers can serve as shared memory between processes: process switch occurs without full register saving and restoring.

The concept of interprocess communication was introduced before the existence of distributed computing systems. In a distributed environment, it is not difficult to adapt the concept of message passing. If a message is meant to be sent to a remote process, it is packaged by the operating system and sent through the underlying network. Upon the arrival of the message, the receiver side unpacks and processes it. But the shared memory scheme does not seem to fit in a distributed application very well. It is difficult for processes running on different machines to share memory efficiently; thus this solution is seldom proposed.

(2) **Message Passing vs Remote Procedure Call**

The basic building blocks for message-based communication vary from system to system. While the *UNIX* system uses the term *socket*, the other systems may use names such as *port* or *mailbox* to represent a similar thing, that is a place to deposit a message. Message is the essential unit in such a communication system. A message-based IPC has some obvious advantages, e.g., simple and efficient implementation, direct introduction of parallelism (using independent *send* and *receive*). But it tends to make programming tricky.

Remote procedure call (RPC) is defined in [Nelso-81] as "the synchronous language-level transfer of control between processes in disjoint address spaces whose primary communication medium is a narrow channel". The idea of RPC is conceptually simple; it is an extension based on a well-known and well-understood mechanism for transfer of control and data within a program running on a single machine — procedure calls. Two steps are usually involved in an RPC: first a message containing input arguments is sent to a *server*, and then a message containing the reply is sent back. The sender is blocked until a reply is received. Thus, unlike message-passing, RPC does not encourage a frequent exchange of information between processes without the notion of master and slave. On the other hand, RPC approach does provide clean and simple semantics. It also reduces program complexity (and tends to de-emphasize concurrency).

Although operating systems usually provide message-passing primitives for IPC, RPC is a language-level concept; it can be viewed as application-level IPC. It is possible to combine the two approaches; an RPC facility can be built on top of a message-based IPC facility.

Communication between processes in a distributed system has two main purposes. One is the transfer of information, and the other is the transfer of control. Based on [Nelso-81], the implications of the two different approaches can be summarized as in Figure 2.1.

	Message Approach	Procedural Approach
means of IPC	messages sent between entities	procedure calls
control	multiple threads of control	independent single threads, one/process
concurrency control	using independent *send* and *receive*	creating and destroying processes

Figure 2.1: The Comparison of Message and Procedural Approach

2.4. Networks

The previous sections discussed the aspects of communication from an operating system's point of view. To send data from one machine to another, the interprocess communication facility of the operating system kernels involved must interface with the underlying network that connects the machines. The understanding of some basic concepts of networks is important to the design of an operating system.

There are two aspects associated with a network, physical (or pure communication) and logical (or application, including operating systems). As we have looked at the issues in operating systems earlier, the following discussion focuses mainly on the physical characteristics of a network with emphasis on local area networks (LAN)[4].

[4] Local area networks are designed for distributed applications over small geographical areas. They are faster and more reliable than other types of computer networks.

2.4.1. Physical Transmission

Transmission Medium

The physical media for the transmission of information range from open-wire pairs to high-speed satellite links. Although most of the media used in conventional telecommunications could be employed in the construction of LANs, *twisted–wire pairs*, *coaxial cable*, and *fiber optic links* are the most often used media in current LAN implementations.

Transmission Techniques

Two techniques can be used for transmitting signals over a physical communication medium: *baseband* using *digital signalling* and *broadband* using *analog techniques*. Note that the communicating devices attached to a network are digital, regardless of whether the network uses baseband or broadband signalling.

With baseband transmission, the entire channel capacity is used to transmit a single data signal, which is in the form of a discrete pulse of electricity or light. *Time–division multiplexing* (TDM) may be used to allow multiple devices on a network to share the same communication channel.

With broadband transmission, a signal is superimposed on a carrier signal by modulating one or more of the three wave characteristics of the carrier signal: amplitude, frequency, and phase. *Frequency–division multiplexing* (FDM) is often used to divide up the physical transmission medium into multiple channels. When multiple devices share the same channel for data transmission, TDM can be used to divide up access to the channel. Figure 2.2 (from [Marti-89]) illustrates the combination of FDM and TDM.

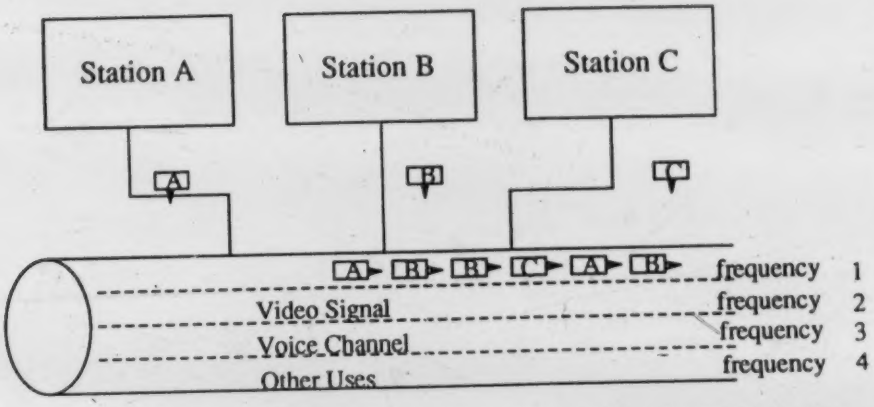

Figure 2.2: Frequency-division Multiplexing with Time-division Multiplexing

2.4.2. Access Control Methods

Access control methods are used to control the sharing of access to the transmission medium. The following is a summary of network topologies (which concerns the physical configuration of the devices and the medium that connects them) and some examples of access control methods employed in a LAN environment.

Network Topologies

Physical (hardware) connection between machines is of one of two types: point-to-point or multipoint.

(1) Point-to-Point

In a point-to-point subnet, two stations (e.g., IMPs) are connected via a channel. Store-and-forward is necessary if the communicating partners are indirectly connected. Some possible topologies for such a subnet are star, ring, tree, complete, intersecting loop, and irregular.

(2) Multipoint

In a multipoint subnet, all nodes share the same communication channel. A message on the channel is received by every node. If a node finds a message irrelevant, it ignores it. Bus, ring, and satellite/radio are a few possible topologies.

For LANs, the *star* , the *bus* and the *ring* are the three most commonly used topologies.

Transmission Control

The transmission control methods can be generally classified as **random control, distributed control,** and **centralized control**. A few examples of different control methods commonly used are presented below.

(1) **CSMA/CD**

CSMA/CD stands for "Carrier Sense Multiple Access with Collision Detection". The access method is most commonly used for LANs that employ a bus or tree topology. It is a random transmission control method. Any station can transmit, and specific permission is not required. Before a station transmits a message, it checks the medium to see if it is free. Occasionally, more than one station sends messages simultaneously, resulting in collisions. To resolve it, each station waits for a random period of time before transmitting again.

The CSMA/CD access method has the advantage of being typically very fast when traffic is not heavy. But under heavier traffic load, the performance may deteriorate due to the increased number of collisions and the time spent responding to collisions and retransmitting.

(2) **Slotted Ring**

Slotted ring is another random access control method used. As the name suggests, it is suited to a network that uses a ring topology. The ring is slotted into a number of fixed

packets; each packet slot has a marker that tells whether it is full or empty. If a station has a message to transmit, it waits for an empty slot, marks it as full, and puts its data in the slot.

The slotted-ring technique is relatively simple. It works well for short messages, i.e., the message packets are small enough to fit in a packet slot. For messages requiring multiple slots to transmit, the technique may not be efficient because of the increase in the overhead for addressing and control information.

(3) **Token Passing**

Token passing can be used on a network that has a ring, bus, or tree topology. It uses distributed transmission control; all stations on the network co-operate in controlling access to the transmission medium. With the token passing technique, a token (a small message) is circulated among the stations connected to the network. Monitoring is necessary to detect problems with the token. With the *token ring* method, a station receives a token marked as free can transmit messages[5]. The token is then marked as busy and appended to the messages. The sender is responsible for resetting the token from busy to free after the token comes back. With the *token bus* method (which is used with both a bus or a tree topology), the station that has the token can transmit messages. The token is then sent to the next station.

The token method has the advantage of allowing greater control over transmission among the stations on the network. But the complexity and overhead involved in token processing and monitoring is high.

(4) **Polling**

---

[5]There usually is a time limit on how long a station can continue transmitting messages.

This is an example of an access method that uses centralized control. One station on the network is designed as the *master* station; it polls messages from all other stations one at a time. It also relays the message from the sender to the receiver. It is most commonly used with a star topology with the central station serving as the master station which is more complex and more powerful.

The polling method has the advantage of flexibility since it allows a station to send a number of messages before the next station is polled. It also makes priority message transmission simple to implement. But the master station becomes heavily loaded, which may cause performance problems. Further more, it creates a single point of failure; if it fails, the entire network fails.

2.4.3. Network Architectures and Communication Standards

Over the years, a number of standards for network architectures have been proposed by standards organizations (e.g., CCITT, ISO, and IEEE), common carriers (e.g., AT&T), and computer manufacturers (e.g., IBM with SNA and DEC with DECnet) to define the rules of a network and how the components of a network can interact. The network architectures are designed to achieve the objectives summarized in [Marti-89]: connectivity, modularity, ease of implementation, ease of use, reliability, and ease of modification.

Network Architectures and the ISO OSI Reference Model

A network is usually organized as a series of layers like the ISO (International Standards Organization) OSI (Open Systems Interconnection) reference model. Although the number of layers, and the names and functions of the layers may be different from network to network, the basic approach is the same: to build the layers so that the details of the lower layers are

hidden and the services required by the higher layer are provided. Figure 2.3 shows a few examples of network architectures and their relationship to ISO OSI reference model (from [Tanen-81]).

Corresponding layers on different machines communicate via a predefined protocol. If one process on machine *A* needs to send a message to another process on machine *B*, both machines must be connected by a communication subnet, which is defined in [Tanen-81] as consisting of the lower three layers of ISO OSI reference model, i.e., Physical layer, Data link layer, and Network layer. It is sometimes called **subnet** for short. The message is delivered from the source machine to the destination machine through the communication subnet — the actual data can only be transferred by going through the lowest layer, that is the Physical layer. On the receiving end, the message is received by the Physical layer and gradually moved up to

<table>
<tr><th>Layer</th><th>ISO</th><th>ARPANET</th><th>SNA</th><th>DECNET</th></tr>
<tr><td>7</td><td>Application</td><td>User</td><td>End user</td><td rowspan="2">Application</td></tr>
<tr><td>6</td><td>Presentation</td><td>Telnet,FTP</td><td>NAU services</td></tr>
<tr><td>5</td><td>Session</td><td>(none)</td><td>Data flow control
Transmission control</td><td>(none)</td></tr>
<tr><td>4</td><td>Transport</td><td>Host-host</td><td rowspan="2">Path control</td><td>Network services</td></tr>
<tr><td>3</td><td>Network</td><td>Source to Destination IMP</td><td>Transport</td></tr>
<tr><td>2</td><td>Data link</td><td rowspan="2">IMP-IMP</td><td>Data link control</td><td>Data link control</td></tr>
<tr><td>1</td><td>Physical</td><td>Physical</td><td>Physical</td><td>Physical</td></tr>
</table>

Figure 2.3: The Network Architectures

higher network layers. Eventually, the message is received by the destination process.

The general functions performed by each of the seven layers in the OSI model are as follows[Marti-89,Tanen-81]:

(1) The *physical* layer is responsible for the transmission of bit streams across a particular transmission medium. The design issues largely deal with mechanical, electrical and procedural interface to the subnet.

(2) The *data link* layer is to provide reliable data transmission with the error-free transmission of data and to shield higher layers from any concerns about the physical transmission medium.

(3) The *network* layer controls the operation of the subnet. It is concerned with routing data from one network node to another.

(4) The *transport* layer determines the type of service (e.g., virtual circuit, broadcasting) to provide to higher layers. It is responsible for providing data transfer between two users at a level of quality agreed upon .

(5) The *session* layer takes the bare bones bit-for-bit communication service offered by the transport layer and adds to it application-oriented functions. It focuses on providing services used to organize and synchronize the dialogue that takes place between users and to manage the data exchange.

(6) The *presentation* layer is responsible for the presentation of information in a way that is meaningful to the network users. This may include character code translation, data conversion, or data compression and expansion.

(7) The *application* layer provides a means for application processes to access the system interconnection facilities in order to exchange information.

Local Area Network (LAN) and 802 Standards

ANSI/IEEE 802 Standards are one set of standards[IEEE-84] for local area networks.

The family of 802 standards deals with the physical and data link layers as defined by the ISO OSI reference model. Figure 2.4 shows the relationship.

MAC stands for "Medium Access Control". There are separate standards describing each medium access method individually. In particular, the access standards define three types of medium access technologies and associated physical media:[6]

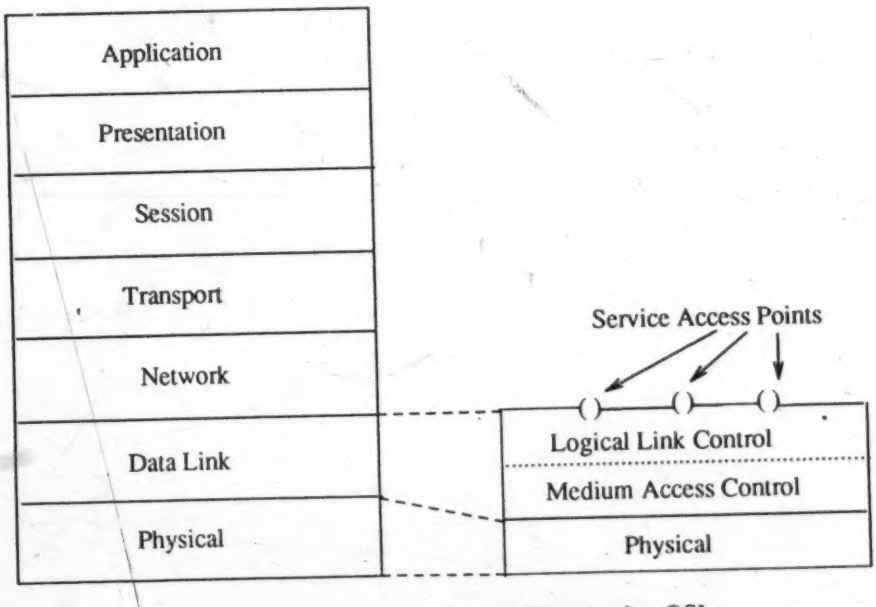

Figure 2.4: 802 Protocol Layers Compared to OSI

802.3: a bus utilizing CSMA/CD as the access method,

802.4: a bus utilizing token passing as the access method,

802.5: a ring utilizing token passing as the access method.

The service specification for the interface between LLC and MAC provides a description of the services that the LLC layer requires of the MAC layer (or that the MAC layer provides to the LLC layer). The services are defined to be independent of the form of the medium access methodology.

LLC stands for "Logical Link Control". This layer constitutes the top layer in the Data Link Control Layer of the LAN reference model and is common to the various medium access methods that are defined and supported by the IEEE/Std 802 activity. The LLC layer interface service specifications to the network layer, to the MAC layer, and to the LLC layer management function are all described by the **802.2** standard.

The 802.2 standard identifies three LLC services[Stall-87].

- **Unacknowledged Connectionless** service, which is a datagram service simply allowing for sending and receiving frames.
- **Connection-oriented** service, which provides a virtual-circuit-style connection between service access points with flow control, sequencing, and error recovery.
- **Acknowledged Connectionless** service, which is also a connectionless service, but provides for acknowledgment.

---

[6]Some other access methods are under investigation.

When an operating system is designed to interface with the network based on the 802 standards, it assumes that there is LLC layer support. Thus, the features of a particular network become unimportant. The network is hidden from the operating system.

2.5. Review of Distributed Operating Systems

There have been many research projects carried out in attempt to investigate distributed operating systems. In this section we review the relevant features of some of these systems that have commonalities with **FLEX**.

2.5.1. Capability-based, Object-oriented Systems

FLEX is designed as a capability-based object-oriented operating system; the prototype of it is to run on top of *UNIX*. Some systems that have these functionalities are **Amoeba**, **Eden**, and **Hydra**.

Amoeba

Amoeba [MulTa-84,MulTa-85,TanMu-81] is an operating system developed for a distributed computing environment at the Vrije Universiteit in Amsterdam. It is a capability-based, object-oriented distributed operating system.

The design of Amoeba is based on the object model in which the system deals with objects, each of which has some set of operations that can be performed on it. The basic paradigm of Amoeba is a **service**. Every object within a certain service can only be accessed through one of the **servers**, the **processes** that constitute the service. Each server accepts requests in the form of **messages** from a **port** on a client process to a **port** on the server process. The server then sends a reply from its port to the client port. Clients can manipulate

objects only via a set of allowed messages. The basic components of Amoeba are processes, messages, and ports.

Protection in Amoeba is based on capabilities to ports. A capability in the system is a bit string that gives the holder permission to use some services, that is to perform some set of operations on some object managed by a certain server.

Communication in Amoeba consists of message-passing between objects via their ports using addresses supplied by capabilities. The communication protocol used by Ameoba is implemented in four layers, instead of the seven (proposed for the ISO OSI reference model)[MulTa-85]:

(1) the bottom layer (the physical layer), which deals with the network hardware,

(2) the port layer, which is responsible for the transmission of 32K byte datagrams,

(3) the transaction layer, which guarantees the arrival of requests and replies[7], and

(4) the final layer which takes care of the semantics of the requests and replies.

To achieve *simplicity*, the kernel uses a pure datagram facility as its interprocess communication mechanism. Note that although most distributed systems have a connection mechanism, Amoeba has joined the trend towards connectionless interprocess communication services[MulTa-85]. The main function of the transaction layer[8] is to provide an end-to-end *message* service built on top of the underlying *datagram* service, with the main difference being that the former uses timers and acknowledgements to guarantee delivery whereas the latter does not.

---

[7]It can be eliminated by the use of *Acknowledged Connectionless Service* provided by 802.2.

[8]A transaction in Amoeba is a request-reply pair.

The concept of **tasks** is provided in the Amoeba system to handle multiple requests from different clients simultaneously without introducing non-blocking transaction primitives. A number of tasks in one address space form a **cluster**; there are specific rules governing the scheduling of tasks within a cluster. As a result, a server can be constructed as a collection of co-operating tasks, each of which handles one request. Non-blocking message transaction primitives become unnecessary. Note that the terms *task* and *cluster* used in Amoeba are similar to *light–weight process* (or *thread*) and *heavy–weight process* (or *task*) used in other systems.

Eden

Eden [AlBLN-85,Black-85,LLAFFV-81] is a capability-based object-oriented operating system developed at the University of Washington. The emphasis in Eden design is on exploring an object-based approach for building distributed applications. Communication is a secondary issue.

An Eden **object** (Eject) has a system-wide unique name, a representation (data part) including a long-term state representing the data encapsulated by the object and a short-term state consisting of the local data of invocations currently in progress, a collection of procedures defining the operations on the object shared by the objects of the same class (Eden-type), and some number of invocations, which are the only means by which one object obtains services from another. Objects refer to one another using **capabilities**, which contain both unique names (object identifiers) and access rights.

Eden objects exist in two possible states: active and passive. An active object has one or more processes executing within the virtual memory of one machine and can initiate activities

as well as respond to messages (which are invocations). In addition, an active object may also contain processes that perform internal housekeeping operations. In particular, an active object can be viewed as a tree structure of processes with the root being a coordinator process. The coordinator process consists of kernel code responsible for maintenance of the object, reception of invocation requests and responses, verification of rights, and dispatching of processes to invocations. A passive object has no processes; it is a result of being deactivated or being forced to crash. Passive objects can be reactivated.

The implementation of Eden is built on top of Berkeley 4.2 *UNIX* [Black-85]; processes, virtual address spaces, access to the disk and network, etc. are all provided by *UNIX*. The Eden kernel on each machine is implemented as a *UNIX* process; Eden objects and the kernel process communicate via *UNIX* IPC. The communication of Eden kernel processes on different machines is done via a simple datagram protocol. Higher level TCP/IP protocols are used only for remote transfer of executable Edentype image files and passive representation. It has been concluded that the high overhead of communication in Eden severely limits its performance (it is one of the disadvantages of building it on top of an existing system)[AIBLN-85]. Although the prototype of Eden is implemented using the facilities of *UNIX*, it does provide users with a complete environment for program development and execution.

The Eden Programming Language (EPL) was developed to give the programmer the illusion of multiple threads of control within each Eject, and to make an invocation look like a procedure call[9]. It provides direct support for the fundamental abstractions (capabilities and invocations) of Eden. It makes the use of Eden significantly easier.

---

[9]It automatically generates stubs for both the caller and the callee.

Hydra

Hydra[WCCJLPP-74,WuLeH-81] is an operating system for C.mmp, the Carnegie-Mellon Multi-Mini-Processor. It was one of the first object-oriented operating systems. The initial design of **FLEX** was strongly influenced by it.

In Hydra, an **object** is an instance of a resource, whether physical or virtual. It can be thought of as a triple:

< unique-name, type, representation >

where the *unique-name* identifies the object, the *type* determines the nature of the resource represented by the object (indicating which operations may be applied to it), and the *representation* contains the actual information content of the object. Protection is achieved using **capabilities**, which are pairs

< unique-name, allowed-rights >.

A capability identifies an object and a list of operations that can be applied to the object.

The message system of Hydra is based on the following concepts:

(1) **port**: Messages are sent between objects of type port. A port represents a service and can be shared. Each port has a set of *output channels*, *input channels*, *message slots*, and a *blocked process queue*.

(2) **connections**: An output channel of one port may be linked to an input channel of another port to form a connection along which messages may travel. The relationship of the connection between output and input channels is one-to-many. Thus, once a connection is established, it is not necessary to specify the destination of a message; an indication of the output channel is all it needs. Input channels can receive and queue messages from

the output channels connected to them.

(3) **messages**: A message consists of three parts, a *text/capability buffer*, a *message type*, and a stack of *reply frames*. It can be thought of as a real piece of storage that can be passed from port to port and can be read and written.

(4) **replies**: The reply stack in a message is central to the message reply mechanism. Whenever a message is sent, a new reply frame with information about the sender is pushed on the message's stack. A reply operation pops off the top frame and uses its contents to determine the return destination. It is possible for a message to be forwarded many times and still return to each point along the inverse route.

A network control program (NCP) is built to interface with **ARPANET**[WuLeH-81]. It serves as a connection manager providing mechanisms for establishing and breaking connections, creating and destroying the sockets that define the participants in a conversation, and transmitting data over connections.

2.5.2. Process-based Systems

Various distributed operating systems are implemented without supporting the concept of user-level objects. Some of the features associated with them also give clues to the design of **FLEX**.

Accent and Mach

Mach [BBBCGRTY-87,JonRa-86,Rashi-86,Sanso-88] is the successor of Accent[RasRo-81]; both are examples of message-based operating systems, designed for distributed systems. The research projects were carried out at Carnegie Mellon University.

There are five objects types supported by the Mach kernel, **tasks, threads, ports, messages,** and **memory objects**. The communication facility of the system is capability-based.

The term process is not used in Mach. Instead, the concept of **task** and **thread** is introduced. A task is the basic unit of resource allocation. Its address space is represented by an ordered collection of **memory objects**. A thread is the basic unit of computation within a task. All threads within a task share access to all of the task's resources. A traditional process in the *UNIX* sense can be seen as a task with a single thread of control. Higher level utility programs can be provided (e.g., C Thread Package [CooDr-88]) to interface the low-level, language-independent primitives for manipulating threads of control so that parallel programming under Mach operating system is supported. **Matchmaker**[JonRa-86] is another programming language support developed for specifying and automating the generation of multilingual interprocess communication interfaces for distributed, object-oriented programming in Mach.

Message and **port** are the basic abstractions used for communication in Mach. Each task has a port in which a message can be placed and later removed. Ports are protected kernel objects. Only the tasks that have a capability (which contains both a port identification and access rights) can send a message to the port identified. A network server task exists on each node of the Mach distributed system to provide transparent network communication. It acts as the local representative for tasks on remote nodes as depicted in Figure 2.5.

Message-passing is the primary means of communication both among tasks and between tasks and the operating system kernel itself. The only functions implemented by system traps are those directly concerned with message communication, all the rest are implemented by messages to a task's port; *send* and *receive* are the only primitive operations. All data passed

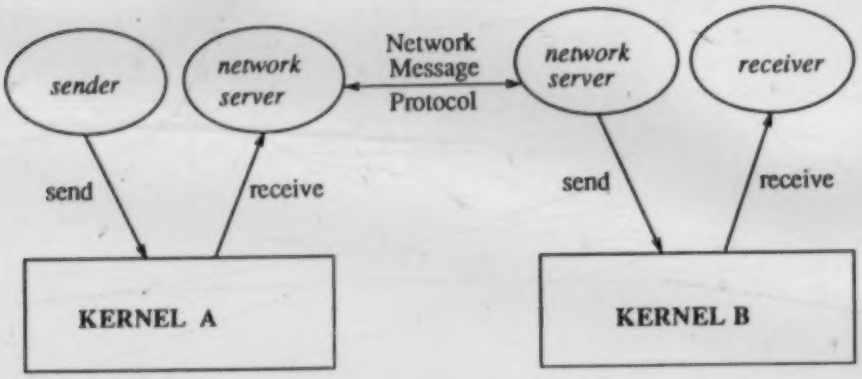

Figure 2.5: Remote Communication in Mach

in a message is typed. All messages have a standard format: a (standard) header with a variable size message body.

Virtual memory management can be used in facilitating interprocess communication as described in [FitRa-86] for Accent. Data between processes can be transferred by integrating copy-on-write virtual memory management with interprocess communication. In particular, Accent provides a flat, 32-bit, sparsely allocatable, paged virtual address space to each process, and uses **mapping**[10] to transfer large data objects between processes and to provide mapped access to files and other data objects. With the concept of memory object, access to virtual memory is simple in Mach with no arbitrary restrictions on allocation, deallocation and virtual copy operations. It also allows both copy-on-write and read-write sharing[JonRa-86]. The Mach operating system kernel also provides flexible sharing of memory between tasks, so that a child task can share the physical memory of its parent task if it inherits shared access to memory from the parent. The tasks may specify different protection or inheritance for their

[10] i.e, the manipulation of virtual memory data structures.

shared regions[BBBCGRTY-87].

V Kernel

The distributed V kernel[Cheri-84_1,CheZw-83], developed at Standford University, implements transparent message-based communication between processes.

V system uses a global (flat) naming space for specifying processes. It also supports the concept of **process groups**. A group is an entity treated like a single process. It is defined as a set of processes referenced by a single identifier. The concept of a group allows the system to take advantage of LAN broadcast service.

V kernel provides two sets of communication primitives. One is for short messages and the other is for large volume of data transfers. It uses a request-response form of communication. The client process that executes a **Send** operation to a server is suspended until the server completes a **Receive** operation and eventually a **Reply** operation to respond with a reply message back to the client.

Protection policies are not implemented by the V kernel. They are left to the individual control of servers.

The V kernel permits multiple processes per address space; these processes form a **team**. The kernel manages memory as team spaces. A team space disappears when the last process in that team space is terminated. The notion of team provides intra-address space multitasking. In a sense, it is similar to the concept of task and thread in Mach.

An important concern in the implementation of the V kernel is efficiency. Special effort is put into it, e.g.,

(1) Remote operations are implemented directly in the kernel instead of through a process-level network server like in Mach.

(2) The inter-kernel protocol is basically a request-response protocol; it implements reliable message transmission on an unreliable *datagram* service without using an extra layer to implement reliable transport.

2.5.3. Distributed UNIX-like Systems

UNIX was designed initially for a single-machine system; it did not provide primitives for distributed applications. The area of interprocess communication has previously been very weak. Pipes were the only standard mechanism which allowed two processes to communicate (prior to the Berkeley 4BSD facilities). Unfortunately, pipes restrict the two communicating processes being related through a common ancestor; the semantics of pipes make them almost impossible to maintain in a distributed environment.

Many versions of *UNIX* have been developed; *Berkeley UNIX* and *System V* are examples of them. Although they are not distributed operating systems themselves, they do provide convenient building blocks for constructing distributed facilities on top of them. But because of the success of *UNIX*, there are also quite a few systems developed to enhance *UNIX* with distributed facilities[11].

Berkeley UNIX

Interprocess communication is one of the important additions to *UNIX* in the 4.2BSD version; the release of 4.3BSD completes some of the IPC facilities and provides an upward-

[11]Even Mach is binary compatible with the BSD 4.3 distribution of *UNIX*.

compatible interface. A system entity, **socket**, which is an abstract object through which messages are sent and received, has been introduced. High-level facilities can be constructed out of this primitive entity. Sockets permit arbitrary processes to communicate with each other. They also make it very easy to extend interprocess communication to a distributed environment.

To support communication networks that use different protocols, different naming conventions, and different hardware, the concept of *communication domain* is used in *UNIX*, i.e., sockets are created within a communication domain.

All sockets are typed; each type represents a kind of service, e.g., datagram socket or stream socket. Connections need to be established between sockets, either on the same machine or across the network, before data can be transferred. Pipes between any pairs of processes (i.e., processes with different ancestors or processes on different machines), can also be built using sockets. Sockets exist only as long as they are referenced.

Locus

Locus[PopWa-85,WPEKT-83] is an operating system implemented at UCLA. It provides a facility for linking computer systems together so as to give the appearance of a single *UNIX*-like hierarchical file store and the standard Shell command language. It can be extended to provide remote file access. Locus is a completely redesigned operating system rather than a modification of an existing *UNIX* system.

Locus uses a special file access protocol. A file access involves three logical sites: using site (US), storage site (SS), and current synchronization site (CSS). If the file to be accessed by the US is local, then a local call is made. If the file is at a remote site, the access request

must go to the CSS first. The CSS locates the SS. If the SS decides to provide the service, it replies to the CSS. The CSS then sends a message back to the US to notify it the fact. Figure 2.6 gives an example of it.

LOCUS also integrated network communication into *UNIX*-like system in the form of transparent remote file access. It does not attempt to provide applications with general IPC across a network. Instead, an RPC mechanism is used for inter-kernel communication.

Newcastle Connection

The Newcastle Connection[BrMaR-82] makes much use of the existing *UNIX* operating system. A software subsystem is added to a set of standard *UNIX* systems to connect them together. The overall hierarchical system structure conceptually is a simple extension of a *UNIX* file system.

The connection is a layer of software sitting on top of the resident *UNIX* kernel, as shown in Figure 2.7. To the layers above it, it is part of the kernel. To the kernel underneath it, it appears to be the same as a normal user process. The role of the connection layer is to

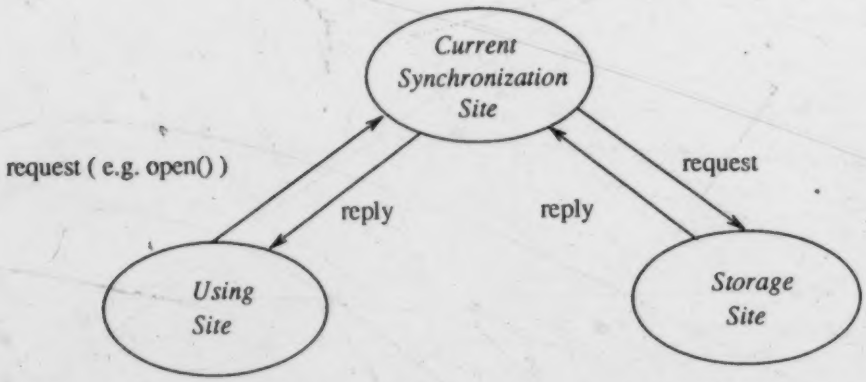

Figure 2.6: An Example of a File Access in LOCUS

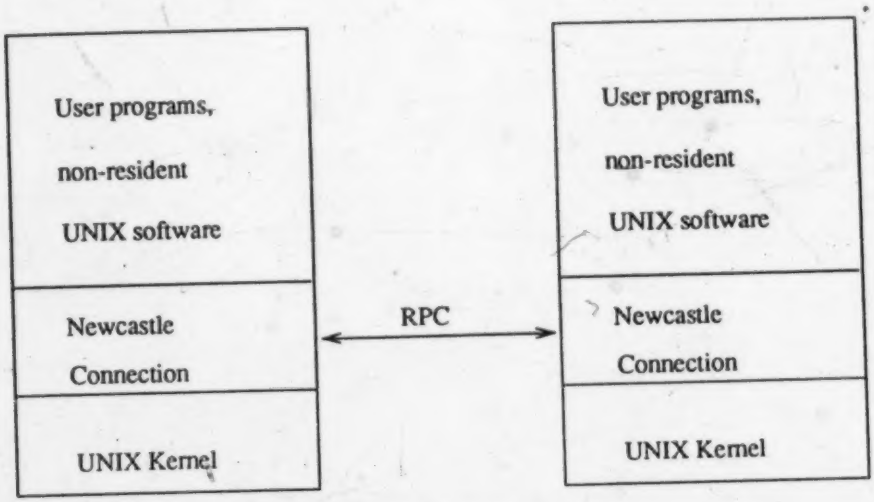

Figure 2.7: The Newcastle Connection

filter the system calls that have to be re-directed to another *UNIX* system, and to accept system calls that have been directed to it from other systems. In other words, if the call is local, it is passed unchanged to the local kernel; if the call is remote, some extra information is packaged and passed to a remote machine. An RPC protocol is used to provide the communication between the connection layers on the various systems. Note that the communication actually occurs at hardware level. The kernel includes means for handling low level communication protocols.

The Newcastle Connection provides network transparency. It is done without any modification to either the *UNIX* operating system or user programs. The idea is applicable to non-*UNIX* distributed systems.

CHAPTER 3

Overview of FLEX

This chapter describes the design of **FLEX**. A more detailed description of the system is given in [OzLLT-88].

3.1. Design Philosophy of FLEX

FLEX is the name of a flexible distributed operating system kernel being investigated. The kernel is developed not only as a testbed, but also as an experiment of an architectural paradigm, so that both architectural and algorithmic issues can be studied. To avoid being restricted to a particular existing operating system, compatibility is not taken into consideration[1]. The **FLEX** kernel should enable us to investigate issues like:

(a) the integration of database management systems (distributed as well as centralized) with operating systems in order to provide better performance and functionality, and

(b) the development of a general-purpose object-oriented computing environment for, among other, distributed object-oriented database managers.

The design philosophy of **FLEX** is discussed in [OzLLT-88]; here is a summary list of it.

Design goals

[1] **FLEX** is not designed to be compatible with any existing operating system.

All (or almost all) operating systems share the same general goals, such as usability, generality, efficiency, flexibility, transparency, integrity, security, reliability, extensibility, etc. While complying better or worse with these general goals, each operating system has its emphasis. For **FLEX**, the following design premises are essential.

(1) The fundamental goal of **FLEX** is to provide the necessary support for distributed database management functions within a general-purpose distributed computing environment.

(2) The **FLEX** kernel should be small and memory-resident for better flexibility and performance.

(3) **FLEX** must provide fully transparent access to system entities, i.e., transparent network access.

(4) **FLEX** architecture must be modular and easily extensible.

(5) **FLEX** is designed to work in a broadcasting local area network environment. Thus, it must conform with the *IEEE 802 LAN standard* at the *Logical Link Control Layer*[IEEE-84] for remote communication.

(6) **FLEX** must provide reliable and secure operation[2].

Design Principles

There are three principles employed in the design of **FLEX**.

Principle 1: The operating system should provide support for only those DBMS functions that can be provided efficiently within the operating system. It should not, however, adversely

[2]However, it is not intended for use in highly sensitive applications where there is malicious intent to compromise the system.

affect the efficient implementation of other services within the DBMS.

Principle 2: Operating system mechanisms should be separated from policy decisions; the kernel should provide the basic mechanisms as basic primitives available to higher levels of software.

Principle 2 gives a way of building a minimal kernel. The kernel implements only the low-level mechanisms that support the policies defined at the user level. This enables us to investigate the issues like the appropriate algorithms for performing the classical database functions within an operating system.

Principle 3: The operating system should support the use of objects at the user-level.

3.2. FLEX Paradigm

FLEX is an attempt to address some of the problems associated with the interface between operating systems and database management systems. Figure 3.1 shows the overall structure of **FLEX**. The kernel is kept small with five modules to provide low-level mechanisms for managing system resources. Services are implemented outside the kernel as user-level objects. In particular, the file system, the scheduler, and the memory manager are placed outside of the kernel. Such an approach allows alternative implementation of services to be made available to user programs. For example, one file server can be implemented specifically for database applications, and another can be implemented to emulate the *UNIX* file system. This also enables us to experiment with different algorithms.

FLEX is not a pure object-oriented system in the sense that the primitive data types such as integers, reals, and characters are not treated as objects. The objects in **FLEX** are typically large and complex. We believe that this decision will significantly improve the performance

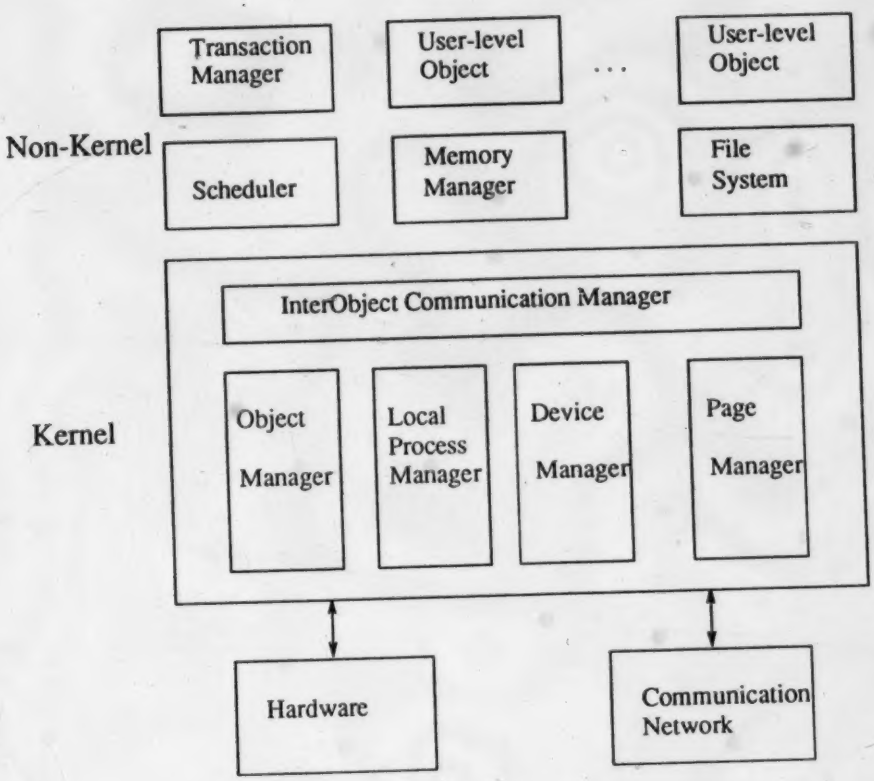

Figure 3.1: FLEX Architecture

of the system because there are fewer objects to manage and less communication for some primitive operations.

The architectural paradigm can be summarized in four points:

(1) Use of a client-server approach to structure operating system modules.

This approach is taken to provide better co-operation between the operating system, the computer network, and the database management system running on top of the operating system. The approach facilitates the use of a dynamic layering approach to operating system design instead of the traditional static layering approach. Layering occurs only during the

execution of the software and is specific to the requirement of that software. For example, if the DBMS does not need the service of a file system, then the file system does not exist as part of the operating system as far as the DBMS is concerned.

This approach fits in a distributed environment very well, especially in an object-oriented system. An (active) object in **FLEX** can be a provider as well as a consumer of services. All objects are prepared to be invoked as well as invoking others. The client and server objects can be distributed to various machines to maintain a balanced load.

(2) Separation of policies from mechanisms.

This principle was first argued for in Hydra[LCCPW-75]. The kernel implements only the basic mechanisms that are necessary for supporting the definition of policies at the user level which govern the allocation of resources. The separation allows the system developers to tune the policies to suit individual applications. For example, there can be one page replacement policy (algorithm) for non-database applications and another for database applications.

This approach makes it possible to implement different policy modules at the user level. As a result, the DBMS can co-exist with the general purpose operating system services — one of the goals in **FLEX** design.

(3) Use of an object-oriented design methodology.

The object model was first investigated in programming languages, but has since been applied to software development such as the design of operating systems[Jones-78,Pasht-82]. Some examples of it have been presented in the previous chapter.

This design methodology simplifies the design of complex systems. It also makes a system easy to extend and modify[Meyer-87,NiBlW-87]. Such an approach is, in a way, the

result of the design decisions of **FLEX** such as the dynamic layering approach and the separation of policies and mechanisms.

(4) Use of capabilities in naming, access control, and protection.

In an object-oriented system, an object, which is the abstracted notion of an arbitrary resource and the only entity known in the system, is the unit of protection. Using capabilities is one way of providing naming, access control, and protection. Reliability of a system is improved because objects can not corrupt one another. The only way to manipulate an object is through a set of operations defined for the object. The name of the object and the desired operation are only identified by an appropriate capability.

3.3. Kernel Design

The most important features of **FLEX** kernel can be summarized as follows:

- The object types supported by the kernel are commonly needed in managing centralized and distributed computer systems as well as in distributed database management.
- A functionally identical kernel exists on each machine.
- Access to objects is uniform: both kernel and non-kernel objects are accessed via capabilities.
- Access to system services and other services provided by user applications is by invocation.
- Message passing is used as the communication paradigm. The InterObject Communication Manager provides fully transparent access to remote objects.

The **FLEX** kernel consists of the following five modules: **Object Manager, InterObject Communication Manager, Local Process Manager, Device Manager,** and **Page Manager**. The general design of each of them is discussed here.

Object Manager

The **Object Manager** provides the mechanisms for creating and deleting objects. It is discussed extensively in [Lau-88].

The **Object Manager** assigns a system-wide unique name to each new object. For each object, the **Object Manager** maintains an **object descriptor (OD)** which contains information about the object. Each site of the distributed system maintains a local object directory which consists of descriptors for each of the objects that exist at that site. The structure of a directory and the manipulation of it will be described in Chapter 4, together with other data structures involved in interobject communication.

A **FLEX** object can be viewed as a triple:

< name, type, representation >

where *name* is a system-wide unique identifier; *type* indicates the class that the object belongs to; and *representation* consists of two parts: one describes the resources that the object encapsulates and the other contains the capabilities that the object holds for other objects.

Each object is an instance of a type which is represented by a type manager. The type manager contains the code of the operations that are defined for that type as part of its representation. Type managers themselves are instances of an object of a type "object" that is implemented inside the kernel. This organization forms a hierarchy of objects whose root is in the kernel. Figure 3.2 shows an example of an object and Figure 3.3 depicts an example of the

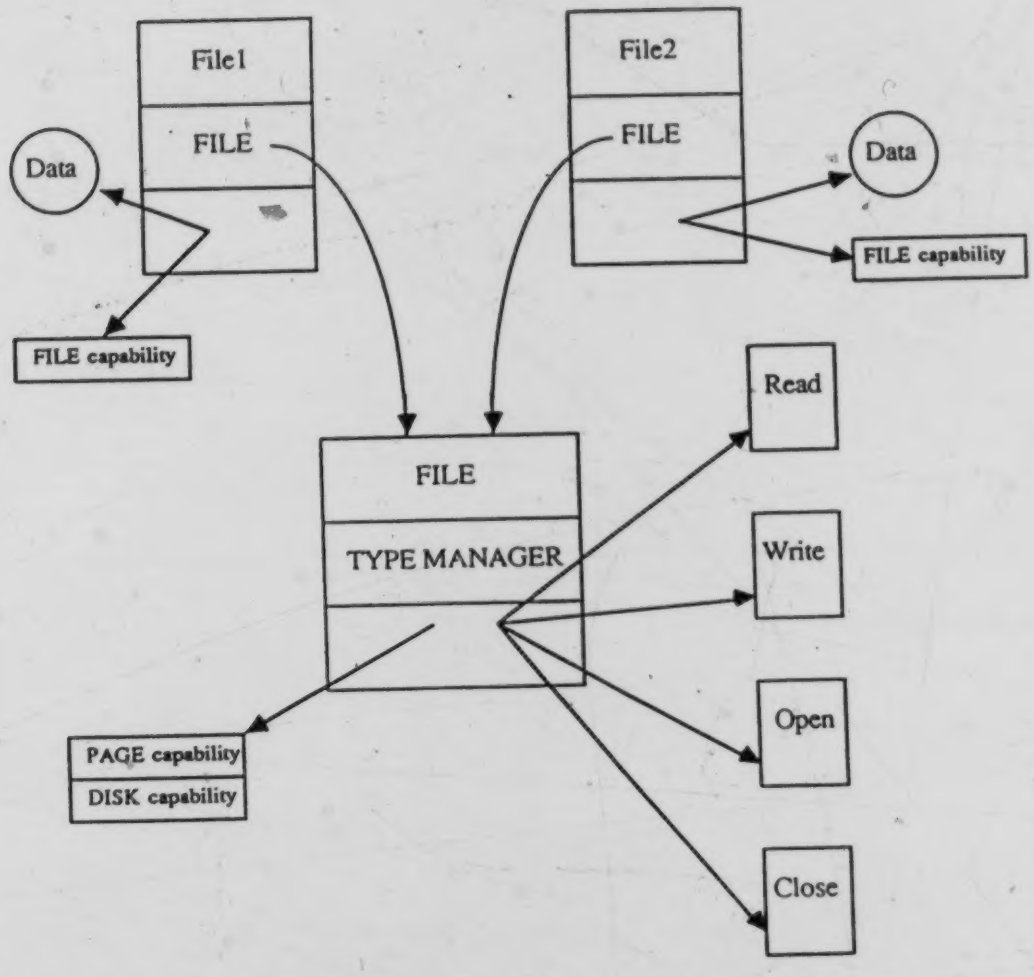

Figure 3.2: An Example of an Object Structure

type (or class) hierarchy.

A **FLEX** object can be either active or passive. Active objects are those that have been invoked for services and are being serviced concurrently by the CPU. An active object may become passive following the termination of all the invoked operations. Each active object is managed by an **object coordinator** (OC) which is a generic piece of code responsible for providing a number of services on behalf of the object. In particular, the OC has the following

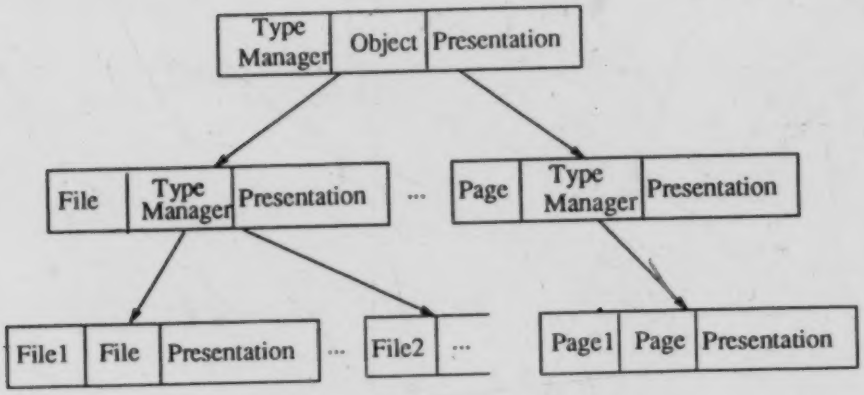

Figure 3.3: Object Hierarchy

responsibilities:

(1) message resolution: to determine from an incoming message which operation is being addressed;

(2) right verification: to determine whether the invoker actually has the right to perform this operation on this object; and

(3) operation activation: to activate the procedure that is being invoked by creating a unit of scheduling and ask the scheduler to schedule it for execution. Figure 3.4 shows the structure of an active object.

In **FLEX**, an object is the unit of protection because it is the abstracted notion of an arbitrary resource and is the only entity known in the system. Access to all objects is done via capabilities. A capability serves two purposes: it identifies the object and it indicates the rights the holder has for the object. When an object is created, a capability is also created. This new capability includes all the access rights defined for the object and is the owner's key to the object. Sharing of an object is done by having multiple capabilities to the object. These capa-

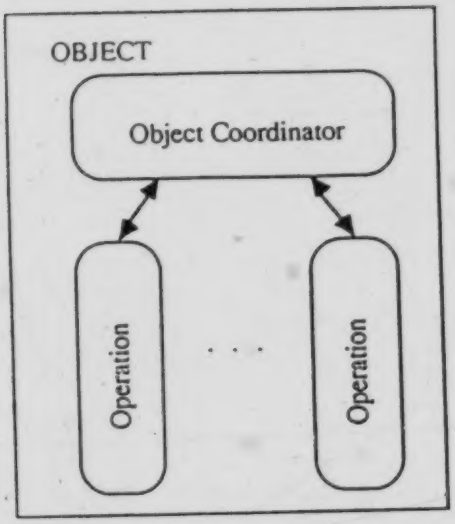

Figure 3.4: An Active Object

bilities might have different set of access rights so that different holders can have different privileges to the object.

InterObject Communication Manager

The **InterObject Communication Manager** in **FLEX** handles all the communication between objects, whether they are local or remote. The details of the design and implementation of communication in **FLEX** are presented in the next chapter; here we highlight the important aspects of the communication subsystem.

The **InterObject Communication Manager** links together all the objects in the system by means of allowing (facilitating) invocations. All invocation calls to an object are, in fact, messages in the system; they must go through the **InterObject Communication Manager**.

FLEX supports both synchronous and asynchronous communication. In synchronous communication, the sender blocks until a reply is received. It is useful for a possible imple-

mentation of remote procedure calls at the programming language level. Asynchronous communication allows the sender to continue operation after a message is sent. A reply may or may not be expected. In any case, the sender will not be blocked. It maximizes parallelism and increases flexibility, but tends to make programming tricky and hard to debug.

The format of a message can be viewed as:

< capability, operation, message >

where *capability* is a token that gives the holder permission to invoke an object; *operation* specifies which routine for the object is to be invoked; and *message* is the parameter list for the routine. A detailed description of how capabilities are used and maintained can be found in [Lau-88]. An outline of how capabilities are used in communication is presented below.

There are two different formats for a **FLEX** capability. One is called **UserCap**, and the other is called **RealCap**. A UserCap is the capability given to the client object, which has a structure as

< object-id, copy-no >

where *object-id* uniquely identifies the object and *copy-no* is a unique integer generated by the system and is used to obtain the corresponding RealCap. A RealCap, on the other hand, is the capability stored in the kernel as an entry in the C-list of the object descriptor. The structure of a RealCap is:

< copy-no, generic rights, auxiliary rights, pointer >

where *copy-no* is the same as the copy-no explained in the UserCap; *generic rights* and *auxiliary rights* are bit maps where each bit defines one operation that can be performed on that object; and *pointer* is for linking up the C-list. Figure 3.5 shows an example of object sharing with different capabilities.

The capability used in a message transfer is a UserCap. The **InterObject Communication Manager** finds the appropriate **OD** of the receiver object using the information provided in the object-id part of the UserCap. Then the C-list of the **OD** is searched to obtain the RealCap with the same copy-no. The rights are then sent together with the message to the receiver. As mentioned earlier, it is the responsibility of the **OC** of the receiver object to enforce the protection of the object by checking the access rights.

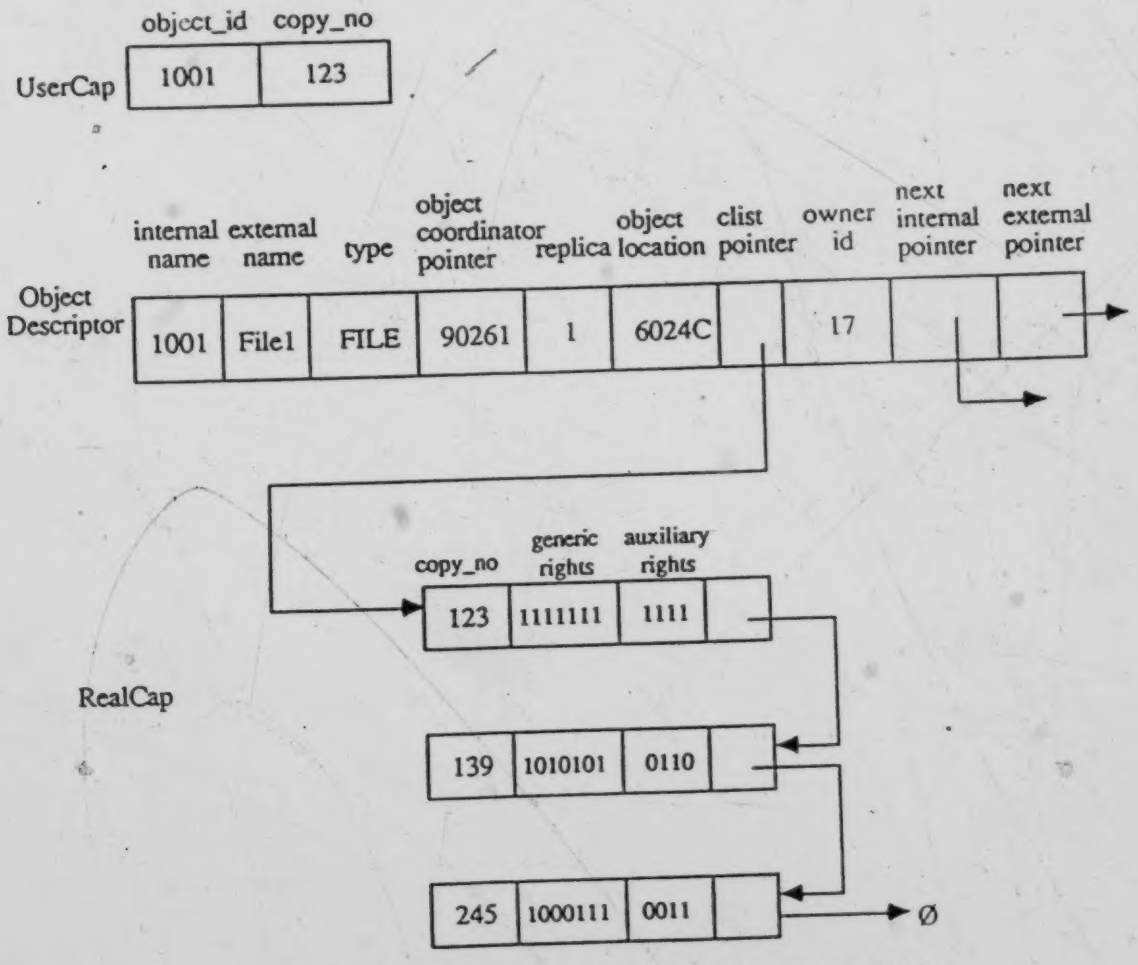

Figure 3.5: An Example of Object Sharing with Different Capabilities

Local Process Manager

The **Local Process Manager** implements the execution environment. A process is a unit that may be scheduled for execution. The **Local Process Manager** performs the following functions related to process and processor management:

- provides mechanism to create and destroy processes,
- performs process dispatching which involves giving control of the CPU to the process that is selected for execution, and
- implements methods to handle deadlocks.

Note that the scheduler makes the decision regarding the execution order of processes; the **Local Process Manager** is simply responsible for context switching.

The operations defined by the **Local Process Manager** can only be invoked by objects that are local to the machine. Remote process creation is not supported. A detailed discussion of it, together with process dispatching, is in [Lau-88].

Device Manager

The **Device Manager** in the **FLEX** kernel consists of all device drivers. A device driver implements the code that is necessary to interface with the device that the driver handles. The hardware of the system is embedded in the **Device Manager** so that device-independent software can be constructed. The device drivers are no different from any other existing operating system.

Page Manager

The existence of the **Page Manager** is the result of the design philosophy: the separation of mechanisms and policies. The **Page Manager** provides the *mechanisms* to support the implementation of a virtual memory system as well as the implementation of various buffer management strategies, both of which are important in implementing database management systems.

It should be pointed out that it is very difficult to enforce the rule of separating policies from mechanisms for this service. Any decision to place some component of the memory and buffer manager outside the kernel has considerable performance implications. In the case of **FLEX**, the minimal facilities for memory and buffer management are provided by the kernel module, the **Page Manager**. The implementation of the policy decisions are left to objects outside of the kernel.

The fundamental operations that the **Page Manager** provides are

(1) to swap in a page identified by its number, and

(2) to swap out a page.

3.4. Non-Kernel Design

The **FLEX** kernel leaves a number of system functions to be implemented outside of the kernel as user objects, which include the **scheduler** and the **memory manager**. The scheduler makes all scheduling decisions and is described in [Lau-88]. The memory manager implements the policies that govern the use of main memory. Some other functions, such as the **file server** and the **user shell**, are also implemented above the kernel.

Little DBMS functionality is implemented in the kernel, but developing a distributed or centralized DBMS on top of the **FLEX** kernel should not be very difficult. The kernel

provides essential services, such as transparent access to distributed objects, mechanisms for manipulating memory pages, uniform naming of objects, and access control facilities. Many operating system functions are pulled out of the kernel. This will not inconvenience the DBMS. For example, the DBMS can implement its own buffer manager that is independent of the operating system memory manager. The suitability of the **FLEX** for database applications is yet to be verified.

CHAPTER 4

Design of the Communication Module in FLEX

Based on the general design of **FLEX** presented in the previous chapter, this chapter describes the detailed design of the communication module in the system.

4.1. Communication Scheme Adopted in FLEX

FLEX adopts a unified approach, adopting active communication, as described in Chapter 2, as its basic communication scheme. Whenever an operation wants to have another operation performed on an object, it issues a request to the corresponding **OC**, which is a daemon. This request is first passed to the kernel via a *svc*; subsequently, the kernel interrupts the daemon and forces it to accept the request. The daemon sees the request as a software interrupt; it can not avoid accepting this interrupt.

It is tempting to demand that the daemon perform the request immediately (without relinquishing the CPU). This would simplify programming: every operation on any object would be atomic. Unfortunately, many applications require that blocks of data be passed around as a result of executing operations on objects (e.g., **copy** a record of a file). Others may require synchronization operations (e.g., a **P** on a semaphore). As a result, they may also want to exchange messages with other operations. In either case (and many others), it is not possible to perform the demanded action immediately; in extreme cases, such as a deadlock, it may never become possible to do it.

Requests to an OC daemon can be trivial, which do not cause inconvenience. But non-trivial requests may introduce a source of concurrency problems: several operations may execute concurrently on the same object; moreover, they may be stacked even before they start their execution. The effects of pre-execution stacking may be seen in the following example:

Code of an operation:	Action performed
i++ ;	submit request: ⟨ *i* , **add** , 1 ⟩
i *= 2 ;	submit request: ⟨ *i* , **multiply** , 2 ⟩

If these operations are considered non-trivial and the original value of *i* is 1, the final value of *i* will be either 3 or 4, depending on scheduling.

At present, the full design of the OCs is not yet complete. Among many responsibilities, an OC daemon performs one of the following once a request is forced upon it:

- processes it immediately (if it is a trivial request) and releases the CPU (goes back to sleep).
- starts a new operation on the object and goes back to sleep. This new operation executes concurrently with all the other operations operating on the same object as well as concurrently with the requesting operation.
- discovers that the request actually is a message to an operation in progress In such a case, the OC places the message in a message queue and goes back to sleep.

4.2. Design Issues

4.2.1. Message Queues

There is no passive communication directed to an OC. But in reality, passive communication still exists. When a message is to be sent from one operation to another operation, the

receiver's OC needs to provide a means of storing the message because the receiver operation may or may not be ready to process the new message.

Passive communication is polling-driven. An episode $P \rightarrow Q$ can be completed only when Q expresses interest in hearing something from either **anybody** or, perhaps, just from P. The most reasonable implementation of polling-driven communication uses message queues to prevent losing messages when the speed of the sender exceeds the speed of the receiver.

For every request, the receiver is represented by an OC, whether the request is an invocation to the OC asking for a service or a data message to a particular operation as a result of a service. OCs are daemons and they can be expected to take care of the messages addressed to them (i.e., creating a new operation) without any serious delays. On the other hand, operations on objects can cause arbitrarily long delays, as they may never ask for a message. Therefore, OCs have to have a storage facility for pending messages which can not be "forwarded" from the OC. The term **supermailbox** is used to denote this storage facility, which is a queue for storing the messages in the OC. The OC is the owner of the supermailbox. Note that a *supermailbox* differs from a *mailbox* as used in operating systems terminology: a *supermailbox* contains messages that are *in transit*, i.e., the OC is not their receiver.

4.2.2. Synchronization

Cooperating operations have the choice of conducting their communication in a synchronous or asynchronous manner. The system provides two versions of system calls — a blocking call for synchronous communication and a non-blocking one for asynchronous communication. It is up to the user objects to coordinate their activities in an orderly fashion.

(1) **Nonblocking-send**: the call returns after the message is deposited in the supermailbox of the receiver;

Blocking-send: the call returns after some sort of reply has been received.

(2) **Nonblocking-receive**: the call returns after an attempt to receive a message from the OC of the object, whether it is successful or not;

Blocking-receive: the call returns after obtaining a message successfully.

For any blocking primitive, a timeout can be set so that a process will never be blocked indefinitely.

4.2.3. Reliability

One of the basic criteria of **FLEX** is to provide reliable message passing. Once a message is posted, it is expected to be delivered. This is not always feasible, though. Therefore, the following definition is adopted:

> Communication is called **reliable** when the kernel guarantees that a message, once sent, arrives safely to the destination supermailbox. Note that a reliable communication does not guarantee that the receiver will read the message, but only that the message will be forwarded to the appropriate supermailbox. The sender may choose to enforce an added degree of reliability by issuing a **blocking-send**.

4.2.4. Protection Enforcement

The protection of objects is enforced by capabilities. This involves the **InterObject Communication Manager** and the receiver's OC. The **InterObject Communication Manager** checks the authenticity of the capability and obtains (from the **object descriptor**)

both the access rights (the *key*) associated with it and the address of the OC where the message is to be buffered, and then the receiver's OC checks the key against the information about the local operations stored in its address space (the *lock*). Only when a matching pair of key and lock is found, the operation specified can be carried out.

4.3. Network Interface in FLEX

In a network environment, successful message transfer can not always be guaranteed. There is always a chance that some site is down. A *reliable* message transfer does not deal with this. It guarantees that no messages are lost and all the messages received are in the right order.

FLEX needs a reliable message passing service. To guarantee it, **Connection-oriented** service (802 *standards*) provided by the network should be employed. But unfortunately, the data link level connections are only for *point-to-point* data transfer between *link layer service access points*. That means that it does not support a reliable **broadcast** feature, as required by the system. To broadcast in a *connection-oriented* service mode implies establishing connections with everybody, which certainly is not desirable.

To solve the problem, either **Unacknowledged Connectionless** service or **Acknowledged Connectionless** service (802 *standards*) may be used. But because of the restriction associated with the *acknowledged connectionless* service, i.e., "[t]he requesting user cannot present a second packet to the link layer for delivery until the previous one is either successfully delivered or determined to be undeliverable"[Jones-88], the *unacknowledged connectionless* service is the only one used in the current implementation of the **FLEX** kernel. It provides the means by which network entities can exchange link service data units without the

establishment of a data link level connection. The data transfer can be point-to-point, multicast, or broadcast.

Although the degree of reliability is compromised with such a decision, the gains are very desirable, i.e.,

(1) It improves the *efficiency* of the system, as there is no need for establishing, using, resetting, and terminating data link layer connections.

(2) It helps in achieving one of the goals in an operating system design, *simplicity*.

(3) It also provides higher degree of *flexibility* to users. If necessary, reliable transmission can be guaranteed by higher lever protocols, e.g., a transport layer above the kernel. Note that it is also possible to make use of the acknowledged connectionless service instead of building a transport layer.

4.4. Classification of Communication Partnerships

The communication partnerships in **FLEX** are illustrated in Figure 4.1. κ's are operating system kernels running on different machines; α, β, and γ are **OC**s representing different objects; P's, Q's, and R's are individual operations running on objects α, β, and γ respectively. α and β are on the same machine; γ represents a remote object.

The possible communication scenarios can be classified as the following:

(1) $P_1 \cdots P_2$: communication between operations belonging to the same object.

In the existing design, the system does not differentiate whether the receiver shares the same **OC** with the sender; a message is always sent to the **OC** of the receiver first. Thus, $P_1 \cdots P_2$ can be decomposed as $P_1 \Rightarrow \alpha \rightarrow P_2$; scenario (2) gives a more detailed description.

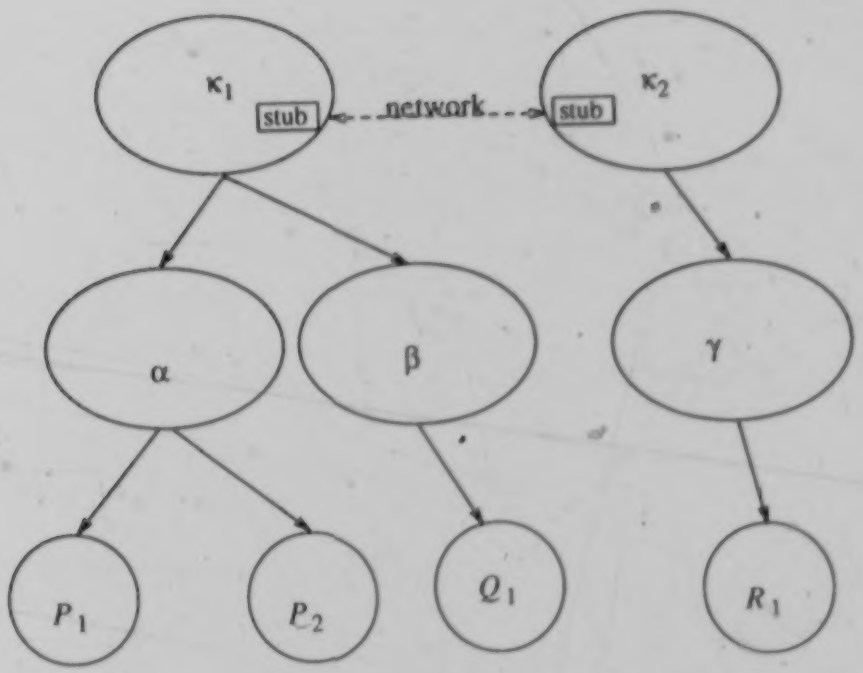

Figure 4.1: Communication Partnerships in **FLEX**

(2) $P_1 \cdots \rightarrow Q_1$: communication between operations on different objects, but within the same machine.

Two steps, i.e., $P_1 \Rightarrow \beta \rightarrow Q_1$, are involved in sending a message from P_1 to Q_1. One is to pass the message from P_1 to β after going through the kernel to obtain the necessary information based on the UserCap supplied with the message, and to store the message in β's address space after having found a pair of matching key and lock in β; the other is to poll the message from β by Q_1. All operations and their OCs are independent scheduling units. Thus, if P_1 sends two messages to Q_1, one immediately following the other: $P_1 \Rightarrow \beta \rightarrow Q_1$, $P_1 \Rightarrow \beta \rightarrow Q_1$, the order of arrival is not necessarily as desired. It is the programmer's responsi-

bility to choose suitable calls (blocking or nonblocking) to synchronize the operations.

(3) $P_1 \cdots \rightarrow R_1$: communication between operations on different machines.

Because of the naming scheme chosen for **FLEX** (see [Lau-88] for details), external or internal names, do not provide any information about the location of objects. Thus, after a message is sent, the local object table (which will be discussed later in the implementation section) is searched first in an attempt to locate the receiver (we expect the majority of communication transactions to be local). If the receiver can not be found locally, remote sites must be checked. The message must first be stored in the local kernel waiting for the stub module to package it and broadcast it via the underlying network. The stub modules on each machine connected to the network are responsible for interpreting and unpacking the incoming message. The message is ignored unless the receiver can be located at that site. The message is treated as a local one there after. The whole message transaction can be viewed as $P_1 \Rightarrow \kappa_1 \rightarrow \kappa_2 \Rightarrow \gamma \rightarrow R_1$.

(4) $P_1 \cdots \rightarrow \beta$: a request to an OC.

A request to an OC implies that the OC is the actual receiver: e.g., a request for starting a new operation, or a request asking for a capability. The OC processes the request immediately. All OCs are daemons; they are always interrupted when a message arrives, i.e., $P_1 \Rightarrow \beta$, whether the OC is the actual receiver or only the receiver's OC.

(5) $P_1 \cdots \rightarrow \kappa_1$: a request by an operation to the kernel that the object belongs to.

It is the same as any other operating system, i.e., $P_1 \Rightarrow \kappa_1$; but the system calls are disguised as messages addressed to a particular kernel module.

(6) Any **intra-kernel** communication is done in the form of procedure calls.

4.5. Semantics of Communication

There are several perspectives on the implementation of message communication:

(1) the user's view of the system,

(2) the function of an OC in message passing,

(3) the network interaction,

4.5.1. The User's View of the System

The user processes do not see the internal implementation of the message-passing facility. To them, the only way to communicate with anybody else (either another process or the kernel) is to use a subset of the following four system calls:

(a) **bsend**(UserCap, operation, msg, size, remote_msg_flag)

(b) **nbsend**(UserCap, operation, msg, size, remote_msg_flag)

These are the blocking and non-blocking **send** primitives mentioned earlier. The arguments of the primitives are the following:

UserCap: the UserCap of the sender; it is used to locate the receiver OC and the access rights to the receiver,

operation: the identifier of the receiver (if the process does not exist, the OC must invoke it by creating a new process),

msg: the content of the actual message to the "operation" (i.e., the parameters for the invocation of the operation or a block of data for the existing operation),

size: the size of "msg", and

remote_msg_flag: a flag indicating whether the message is originated from a remote site.

A call to **bsend** will not return until the entire message transmission is executed successfully; the message is processed and a **reply** is sent back ("nbsend" is used for sending a reply). If the expected reply is not received by the sender within a certain time limit, the delivery of the message is assumed to have failed.

A call to **nbsend** returns when the message has reached the supermailbox of the receiver. No reply can be expected. It is up to the cooperating objects (or their operations) to synchronize their executions. Whether an acknowledgement is required can be indicated in the body of a message. In particular, they may set up their own acknowledgement/timeout protocol.

(c) **breceive**(sender_id, msg, size)

(d) **nbreceive**(sender_id, msg, size)

These are the blocking and non-blocking **receive** primitives. *sender_id* allows the receiver to specify the sender of a message. It is desirable when an operation is communicating with a number of other operations. *msg* and *size* are parameters for the message to be received and its maximum acceptable size.

A call to **breceive** will not return until the expected message is moved from the supermailbox of the OC to the actual receiver's address space.

A call to **nbreceive** returns regardless the presence of a message in the supermailbox. If the expected message is there, the receiver operation accepts it; if not, the receiver operation returns anyway (with an appropriate status).

4.5.2. The Function of an OC in Message Passing

The OCs play an important role in communication. They are responsible not only for providing protection to the objects (the OC has extra responsibility in checking the access rights of the sender), but also for managing their supermailboxes and relaying transit messages to subsequent points. Two types of messages may arrive at an OC:

(1) A transit message, addressed to an operation of the object. The OC stores such a message in its supermailbox and passes it farther at an appropriate time.

(2) An invocation requesting the execution of some action on the object represented by the OC. In such a case, a new process is generated.

The code of OC is provided by the system. The communication primitives used by OCs are not expected to be used by user applications. The primitives are non-blocking and do not guarantee success; the main purpose of their existence is to diminish the risk of deadlock in arbitrary communication sequences.

4.5.3. The Network Interaction

The network is represented as a stub module at each site. The stub modules are implemented as system daemons. Each of them is also a link with one end of it connected to the network software (LLC of 802 standards) and the other end to the kernel of the site where it resides.

CHAPTER 5

Implementation of the Communication Module in FLEX

This chapter describes the implementation of the design of the **FLEX** communication module presented in the previous chapter.

5.1. Implementation Details

The prototype implementation of **FLEX** is done on top of *UNIX*; *UNIX* processes, signals, files, sockets, etc. are used to simulate the environment. Remote communication is also simulated by running the **FLEX** kernel on several machines.

The communication facility implemented is only a part of **FLEX**. At this moment, it does not interface with other parts of the system. Thus, some assumptions are made for the purpose of testing the communication subsystem.

(1) Capabilities are assigned statically; no support exists in the prototype for dynamic creation of capabilities. The *internal names* and *access rights* are simulated.

(2) Operations are all independent processes. Objects are represented by **OC**s, with no real physical or logical realization.

Note that a timeout parameter may be set for each request to avoid indefinite postponement. It can be set up as an option to users or as a default controlled by the system. In the prototype, the problem is not dealt with.

5.1.1. Representation of Senders and Receivers in FLEX

When a nonkernel object is first created, an OC process is generated to represent the object. It manages any operations to be performed on the object which it represents. All the information about the operations on the object is stored in the OC.

An operation on an object can be performed by either a procedure call or an independent process. Both are initiated by the OC according to a request. While a procedure call is executed in the context of the caller, which is the OC in FLEX, a process runs independent of its creator. At any moment, only one operation of each kind can be active; if an operation is addressed by its name, there is no ambiguity. But because of the responsibility of an OC in protection enforcement, a procedure running in its context may jeopardize the security of the system; thus, only an independent process can be used to represent an operation in the prototype implementation of FLEX.

In summary, the sender of a message in FLEX can be any operation represented by a *UNIX* process; a receiver can be either an OC or an operation, both of which are represented by *UNIX* processes.

5.1.2. Representation of Kernel Address Space

The prototype implementation of FLEX runs as a collection of *UNIX* user processes; thus, it can not access real kernel address space. To solve the problem, files are used to simulate kernel address spaces on different machines. The data structures used by the FLEX kernel are all stored in files so that kernel modules can access them.

5.1.3. Representation of Network Support

A stub module in the kernel is responsible for handling remote communication. It is supposed to interface with **Ethernet** according to **IEEE 802 standards**. We take advantage of the remote network communication facility provided by *NFS*. Thus, when the kernel residing on one machine sends a message to another kernel (residing on a different machine), remote file access is used to move the text of the message from the sender's address space (a file) to the receiver's address space (a file on a different machine).

A broadcast is not available to processes using *NFS*. Therefore, it is simulated by sending individual messages to all the machines controlled by the **FLEX** system.

5.1.4. The Data Structures Involved in the FLEX Kernel

(1) Object descriptor

An **object descriptor (OD)** stores information about an object. It is created when an object is created. It is maintained by the kernel. The format of an **OD** is given in Figure 5.1, with the following field definitions:

internal name
external name
type
object coordinator pointer
replica number
object location
c-list pointer
owner id
next internal pointer
next external pointer

Figure 5.1: Structure of an Object Descriptor

internal name: an integer, generated by the kernel, that uniquely identifies the object.

external name: a character string that the user chooses to name the object. It needs not be unique.

type: a character string describing the type of the object[1].

object coordinator pointer: an integer that stores the process identifier for the object coordinator of the object.

replica number: an integer describing the number of existing replicas of the object.

object location: a disk address/pointer to the object. This field is updated if the object is moved to another location.

c-list pointer: a disk address/pointer to the capability list held for the object. The structure of a capability and its manipulation is discussed in [Lau-88].

owner id: an integer that identifies the owner of the object.

next internal pointer: a disk address/pointer to the next OD based on the internal name.

next external pointer: a disk address/pointer to the next OD based on the external name.

(2) **Object table and Local active object table**

All local ODs form a single level directory at each site. It can be managed by using hashing with the help of two tables, which are an **Internal object table (IOT)** and an **External object table (EOT)**. The tables have *internal names* or *external names* as their access keys. Each entry points to a corresponding OD.

---

[1]The string can either be "type manager" or the name of the "type manager" that defines the object type for the object. The detailed description is in [Lau-88].

IOT and EOT can be treated as one entity — **Object table (OT)**, which keeps all the ODs of the objects created at that site. An active object has an entry not only in the local **OT**, which is stored in secondary storage, but also in **Local Active Object Table (LAOT)**, which is a subset of **OT** kept in the main memory all the time for faster access. **LAOT** has the same structure as **OT** and contains objects which are currently active. An object is always created as an active object. It may later become passive because of a scheduling decision or an explicit passive command. When an object becomes passive, it loses its entry in **LAOT**. Of course, when it becomes active again, the opposite happens.

As mentioned above, **OT** and **LAOT** store the information about all objects and active objects respectively. They are constructed as **hash tables**, the size of which is defined at the system initialization time. Colliding entries in the tables are arranged in **linked lists** of ODs. The object manager maintains the tables. The **InterObject Communication Manager** accesses them (at least **LAOT**) for every message transfer.

OT and **LAOT** can be accessed by either **external (user defined)** or **internal (system assigned)** names. While the internal names are unique, the external names are not. The description of the internal organization of the **OT** and **LAOT** tables can be found in [Lau-88].

(3) **Message format**

All messages in the system have the same format as shown in Figure 5.2: a message header plus the body of the message.

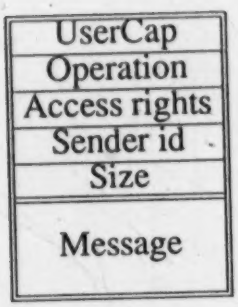

Figure 5.2: Format of a Message

A message header consists of five fields: the capability of the sender (UserCap), the name of the operation in question, the access rights associated with the capability, the identification of the sender, which consists of the internal name of the sender object and the name of the sender operation, and the size of the body of the message. The body of a message can be anything, e.g., parameters for the operation specified or a block of data as a result of an operation.

The access rights field is only filled if the receiver is found locally. The sender's information is provided by the kernel. The rest is the same as what is specified by the sender.

As the format of the header of a message is fixed, it is easy to find out where the body of a message starts. To maintain the uniformity of the message passing system, information which may not be necessary at the final destination is nevertheless provided.

5.1.5. Message Buffering in Supermailboxes

5.1.5.1. Location of Supermailboxes

As the design includes a *non-blocking* send, one must expect that there will be a queue of pending messages. The queue is represented as a supermailbox belonging to the OC of the

receiving object. The choice of where to store the supermailboxes with messages is not obvious, as there are two options:

(a) to store messages in the kernel address space, and

(b) to store messages in the OC's address space.

If the messages are buffered in the kernel, a uniform message buffering scheme for both local and remote communication is achieved. But, as communication with an object is interrupt-driven, the requests to an OC can be taken care of with no significant delay; it is never necessary to buffer those requests. The real necessity of a buffering scheme is for the messages sent to operations. Since operations are seen as part of an object, which is represented by an OC, the OC has to store all the messages for the operations somehow[2]. Thus, there is no reason why a message has to be buffered twice, both in the kernel and in the OC.

If a supermailbox is put in the address space of an OC, the number of times that a message has to be moved is once: from the sender's address space to the address space of the OC that owns the supermailbox. It makes the message passing facility more efficient. But note that it fails in providing a uniform message buffering scheme for local and remote communication; a remote message needs to be stored in the kernel waiting for the stub module to handle it.

For the reasons outlined above, FLEX stores messages (i.e., supermailboxes) within the address space of the corresponding OC.

---

[2]How the messages are going to be passed to the operations from their OCs is "internal" to the objects.

5.1.5.2. Structure of a Supermailbox

The size of a supermailbox is fixed by the system. That means that the number of messages that can be stored in a supermailbox is limited. If there is no space for an incoming message, the sender will be blocked even in a "nonblocking send".

A supermailbox has a slot for each operation running on the object. It also has special slots for the messages that have not been looked at by the OC, and for messages that can not be delivered (e.g., orphan messages). Figure 5.3 outlines the structure of a supermailbox.

Sometimes, a sender sends a message to a process that does not exist at the moment of sending. Such messages, called **orphan messages** can either be discarded as faulty or kept until the receiver materializes. **FLEX** uses the latter approach — this helps avoiding difficult

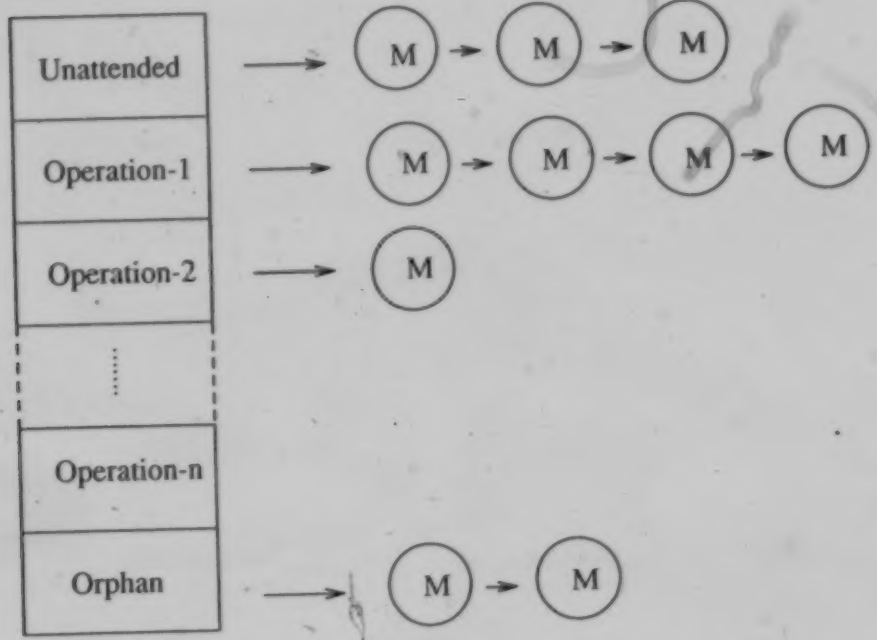

Figure 5.3: Structure of a Supermailbox

synchronization problems involving unpredictable sequences of scheduling decisions. A slot in a supermailbox is provided to buffer orphan messages; it is checked every time a new slot for a new operation is started. A time limit can be set for how long an orphan message is allowed to exist. If the message has to be terminated, the OC will send a message to the sender of the orphan message indicating the status.

5.1.5.3. Mechanics of Message Transmission

There are two ways of depositing a message in a supermailbox. One is to **force-write** the message into the address space of the OC, which does not require any cooperation from the part of the OC; an interrupt (e.g., a signal) can be sent to the OC to notify it of the fact that a message has arrived and is waiting for being processed. The other is to wait until the OC is willing to receive the message. The former corresponds to *active* communication and is the approach taken in **FLEX**.

Note that communication can be combined with virtual memory management; movement of local messages only involves updating page table entries.

5.1.6. Message Movement in FLEX

For each successful message transfer, the following steps may take place:

(1) Locate the OD of the receiver

When a message is trapped by the **InterObject Communication Manager**, the kernel looks at the UserCap sent together with the text of the message and tries to locate the OD of the receiver. This is done by hashing the internal name ("object id" part of the UserCap) into the LAOT of the system; the resulting OD should have all the information necessary for the

message transfer. Note that if the search fails, it implies that the receiver is on a remote site. The message needs to be buffered in the kernel space waiting for the stub module to package it and broadcast it to the network.

(2) Locate the supermailbox and access rights

The identification of the process representing the OC is available in the OD; it gives the information necessary for locating the supermailbox. To obtain the access rights, the c-list of the OD must be searched to find the matching copy of the RealCap using the "copy number" part of the UserCap. If a RealCap is found, it means that the UserCap is authentic. The message should be passed to the OC of the receiver object with the access rights stated in the RealCap.

(3) Process the message by an OC

The message is updated with the access rights from the RealCap. It is put into a special slot of the supermailbox of the OC waiting for the OC to process it. The OC daemon is interrupted so that the message is handled immediately. If the message is for the OC, it is taken care of by the OC and the slot is cleared. If it is for an operation under the care of the OC, the message is moved to the slot that is dedicated to that operation.

(4) Deliver the message to the receiver

This step only occurs if the message is for an operation. The operation polls the message from the supermailbox. It sends a request to its OC, and the OC finds the message from the supermailbox for the operation, the real receiver of the message.

(5) Remote message transfer

If a message has to be sent to a remote site, it will be a **remote message**. In such a case, several stub modules must get involved. The message transfer becomes more complex.

If the **InterObject Communication Manager** can not locate the **OD** successfully, it implies that the receiver object is not local to the sender. Thus, the message needs to be put in the kernel (mailbox) of the sender and be broadcast to all the machines connected to the network. A local stub module working like a daemon waits for an interrupt informing it that a remote message can be found in the kernel mailbox. Then it packages the message properly for the network transmission.

The stub module on each machine connected to the network receives the message. It unpacks it and tries to locate the **OD** using the information provided by the UserCap. If it fails, the message is ignored; if it succeeds, the message is treated as a local one.

5.2. Comments on Implementation

Due to the lack of hardware support, the implementation is simulated using the C programming language running on top of *UNIX*. The purpose of such a simulation is to prove that the prototype of the design works and to get some insight for potential improvement of the design. The simulation code will eventually be moved onto a real machine. Of course, changes have to be made to deal with the real hardware.

The simulation was based on the original design of **FLEX**. *UNIX* is only a tool to facilitate the development of the system; it provides an environment that is much easier to use than a bare machine. Note that the **FLEX** communication facility is conceptually independent of the operating system it currently runs on. For example, the *UNIX IPC* facility (i.e., *socket*) is used to implement intra-object communication (polling) and remote file access is used to

simulate remote message transfer. Also note that some of the design features can not be simulated accurately on top of *UNIX* because *UNIX* does not support the virtual machine concept.

CHAPTER 6

Recommendations for Improvement

The previous chapters presented the first version of **FLEX**, especially its design and implementation in communication. As the understanding of the issues involved became better during the work on this thesis, it led us to some recommendations for improvement in the next version of the **FLEX** design.

This chapter highlights some major changes which may be applied to the first version of **FLEX** as possible improvements for the next version. It then proposes a basic design of a new version of **FLEX** with the changes, and an overview of communication in the new design.

6.1. Major Changes to the Previous Design

Although some changes are recommended for the design of **FLEX**, overall design goals and principles remain the same. As **FLEX** is an ongoing research project, more changes are expected in future work.

6.1.1. Capability Concept

The term capability was used in presenting the system earlier. But a close look at it shows that the capability discussed earlier is actually a hybrid of the concepts of capability and access rights, which is referred as a lock/key mechanism in [SilPe-88].

In a system that uses access rights as means of protection, an access right list is the property of a service, which can only be modified by "trustworthy" procedures. In the case of

FLEX, object coordinators shouldered the responsibility of maintaining the lists of access rights. Since the code of an object coordinator was provided by the kernel, i.e., it was a generic piece of code and, as long as there were no processes running in the context of the object coordinator and the code was correct, "trustworthyness" was guaranteed.

A capability is like a key to a service. It is owned and stored by a client. If a client has a capability to a particular service, it will get the service unless something extraordinary happens (e.g., server dies in the meantime). The capability specifies the service, and the ownership of a capability guarantees that the service will be provided. There are no access rights involved; in particular, there is no need to check whether a capability is legitimate. The kernel translates a capability supplied by a client into the appropriate operation on the given object. As defined in [Pasht-82], a capability is a pair

$$\langle\, object\,,\, operation\,\rangle.$$

Because of the concept of ownership, capabilities can be duplicated and passed around as the owner wishes. Of course, a detailed design of a capability-based system may be quite complex, e.g., some capabilities may not be allowed to be duplicated.

The UserCap in FLEX played part of this role. The problem was that even after obtaining the RealCap using the UserCap, the success of the operation requested was still not guaranteed. The kernel was responsible for checking the authenticity of a capability, but not the rights for the operation requested.

In the updated version of FLEX, classes and objects can be created dynamically; class hierarchy is supported. A *subclass* can inherit all the properties of its parents, including the ability to access other objects — and this is essentially the same as the notion of capability.

Thus, in the proposed version of the **FLEX** design, the concept of capability (in the strict sense) is employed.

If the capability approach is used, access rights become unnecessary. The responsibility of checking access rights can be removed from object coordinators, which will be redefined later. For the purpose of communication, we assume that a capability identifies the receiver and the operation to be performed. The detailed design of it is left as part of future research.

6.1.2. Object Model

The **FLEX** kernel designed earlier could not support multilevel class hierarchy. Although a class can be defined dynamically, it is always an object of type "Object" as shown in Figure 3.3.

It is essential that **FLEX** provides an environment that allows users to build their own class hierarchy. As a result, they can create their own subsystems if necessary. Thus, two features are added to the kernel:

- Object classes can be created dynamically during the execution of the code of other objects; thus, the concept of **subclass** is supported.
- An operation on an object is the unit of scheduling. The kernel of **FLEX** treats classes in the same way as more traditional operating systems treat (heavy-weight) processes; it treats operations on objects as (light-weight) threads.

6.1.3. Object Coordinator

The concept of object coordinator (**OC**) was introduced in the earlier design. One **OC** was associated with one object. Among other things, it was responsible for access rights

enforcement.

As the result of supporting multilevel class hierarchy and the use of capabilities, the role of an OC is changed.

(1) The code of OCs is not generic — a part of it is defined by the user as part of the application. In other words, users write their own OC code for the classes that they want to create.

(2) There is one OC for each object class instead of each object.

(3) An OC represents a class. It keeps the information about the objects of its class; it also keeps some context information about operations in progress (threads).

(4) Whenever there is a need to either create an object or perform an operation on an existing object, an appropriate message has to be sent to the corresponding OC. The OC is responsible for converting these messages into kernel requests and passing them to the **Object Manager** or **Thread Manager**.

(5) It is not possible to send messages to objects directly: they must be sent to their OC instead, as in:

$$\langle\, O\ ,\ \textbf{read}\ ,\ \textit{text of the message}\,\rangle$$

In a way, the OC buffers the messages to the objects of its class. This scheme allows the use of any sort of message passing in user applications. It also helps to overcome trivial timing problems, e.g., messages[1] to an object that has not been created yet.

---

[1]In the sequel, the term **message** will have two somewhat different meanings: an **operation invocation** addressed to an object coordinator, or a **block of data** addressed to an object.

(6) OCs are responsible for local clean-up after an object is deleted.

(7) OCs do not enforce access rights in a capability based system. The protection is done by the kernel. Note that in this design, a capability represents an unconditional right to start an operation on an object (i.e., a thread).

6.1.4. Object Manager and Local Process Manager

In the proposed version of the **FLEX** design, the responsibilities of the **Object Manager** are changed. Instead of having a **Local Process Manager**, a new kernel module — the **Thread Manager** — is defined. Yet another kernel module, the **Dispatcher**, dispatches threads; besides, it takes care of the kernel side of CPU scheduling.

The kernel must be able to create and delete objects. It also creates, dispatches and terminates light-weight threads.

- The **Object Manager** creates a new object by allocating the appropriate resources to the requesting object coordinator.
- The **Object Manager** is also responsible for global clean-up after an object is deleted.
- **FLEX** treats light-weight threads as units of scheduling. Each thread represents one operation performed on an object of a class. Thread creation is triggered by a request to the **Thread Manager** issued by the object coordinator of this class.
- The **Dispatcher** is not part of the **Thread Manager**; it is an independent module in the kernel. When the CPU becomes available, the **Dispatcher** selects one heavy-weight process and invokes the user-supplied scheduler associated with this process. This scheduler (which resides outside of the kernel) selects one thread belonging to the

heavy-weight process. Based on the decision made by this scheduler, the **Dispatcher** assigns the CPU to the selected thread.

The data structures used by the **Object Manager** (and the way they are manipulated) are still the same, e.g., the OD, the single level directories.

Note that in the future, the **Object Manager** will have to be augmented, so that it can perform more complex operations than those discussed above.

6.2. The Proposed System Design

This section proposes the basic design of **FLEX**, the main emphasis being on objects and object manipulation. The main subject of the thesis, **communication** is only touched and not described in detail here.

6.2.1. The Overall Structure of FLEX

FLEX is an operating system based on the concepts of object-oriented programming.

In programming languages, an object is an instance of an abstract data type (**object class**). It is defined as an entity upon which a predefined set of operations can be performed. While it is not very easy to use the concept of objects in the design of operating systems, especially their kernels, it is convenient to use this approach in the design of the user-level interface. The object concept provides a convenient tool in designing and implementing process management and communication in an operating system, especially from the point of view of protection mechanisms [Jones-78, Pasht-82].

At the heart of **FLEX** is an entity called the **KERNEL**; it is made up of several kernel modules; among them are **Object Manager**, **Thread Manager**, **Dispatcher**, **Page Manager**,

Communication Manager and **Device Drivers**. Kernel modules can access directly the hardware[2] of the system.

Several important system functions are left outside of the kernel of **FLEX**; among them, the **Scheduler** and the **Memory Manager**.

All the other entities in **FLEX** are implemented as **non-kernel object classes**. Some standard classes are created by the kernel automatically at start-up time. They provide some basic facilities to the users and application software:

- a file system,
- the ability to execute a program,
- the ability to communicate with the outside world.

While the standard classes are always present, additional classes can be defined freely by application programs. This can be done in a dynamic fashion, during the execution of these programs.

6.2.1.1. The Kernel

Figure 6.1 shows the kernel structure of **FLEX**.

The basic modules of the kernel are:

- **Object Manager**, responsible for creating and deleting class objects. It also maintains definitions of user-defined classes.
- **Thread Manager**, responsible for creating and terminating units of scheduling (threads).

---

[2]As in any other operating system, the network is simply treated as yet another physical device.

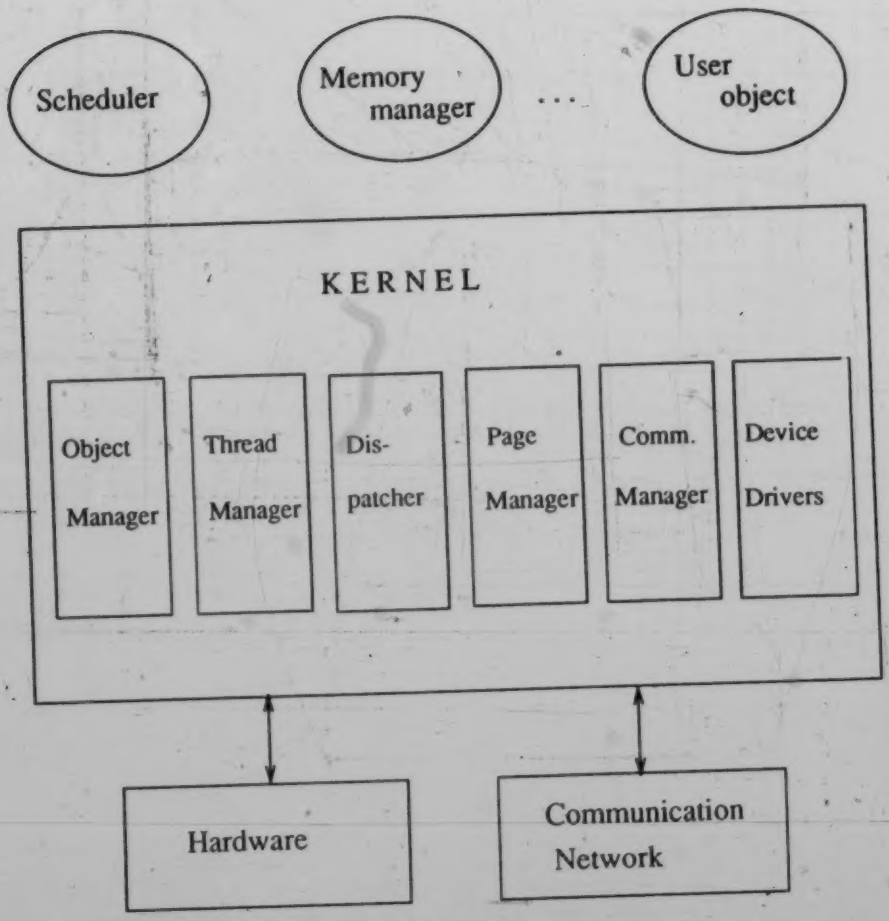

Figure 6.1: **FLEX** Kernel

- **Dispatcher**, responsible for rolling new contexts into the CPU.
- **Page Manager**, responsible for low-level management of virtual and real memory.
- **Communication Manager**, responsible for routing communication items from their senders to their receivers. As a byproduct of this activity, the **Communication Manager** is also responsible for creating capabilities, decoding them and garbage collection of

objects[3].

- **Device Drivers.**

Besides these modules, the kernel contains a software-interrupt handler, which catches all the system-call interrupts (*svcs*).

6.2.1.2. Applications

A user who is interested in using **FLEX** may write a program and submit it for execution. This program may be written in any language that has a resident compiler, but the design of **FLEX** is meant to facilitate the use of object-oriented programming languages. For simplicity, we assume that user application programs are indeed written in object-oriented programming languages. The semantics of these languages will not, however, be taken into account in this discussion.

When an application program is submitted for execution, it contains the definition of at least one class. **FLEX** gives the facilities needed to define object classes and sub-classes within these classes. Classes may be defined at load time or at execution time.

One of the assumptions about **FLEX** is that it will control a distributed computer system. As a result, it should have a good mechanism for concurrent (if not parallel) execution of several operations on one and the same object. Such a mechanism is supplied in the form of **threads**. The presence of such a mechanism has an obvious side effect: it puts the obligation on concurrency control squarely on the shoulders of the application programmer. This is both good and bad, as, on the one hand, it allows the implementation of arbitrary synchronization

---

[3]An object is considered an orphan when no other object has the capability to access it. In such a case, the orphan object should be terminated.

schemes, while on the other hand, it forces application programmers to perform additional programming work (of a kind that they hardly are familiar with). It is believed that the negative aspect can be relieved through the creation of a library of standard synchronization tools that will be made available to applications.

6.2.1.3. Daemons and Threads

The traditional operating system recognizes two types of active entities (capable of using the CPU): processes and daemons. A daemon is similar to a process, but it is in the form of an infinite loop, sleeping in this loop until it is awaken by an interrupt. Traditionally, daemons have been used to implement mainly system servers (such as the *UNIX* pagedaemon).

In traditional systems, when there is a need for several operations performed on a resource concurrently, several processes are generated, each of them independently performing one operation. This is often impossible to implement safely, and more contemporary systems use the concept of a multi-thread process.

The idea is to split the concept of a process into two levels:

- a global entity, containing the whole context of the process, its code and data, etc. This level is called **heavy-weight process**.
- a number of local entities, each of which represents one currently performed operation on the data of the process. Such an entity is called a **light-weight process**, or, in a slightly different jargon, a **thread**. A thread represents a small part of the context of the process: location counter, stack pointer and other registers, and maybe a few other items.

Several threads may concurrently share one heavy-weight process: they represent the concurrent execution of several pieces of code which share the same global data area, but are

independent of each other.

In this approach, every heavy-weight process behaves in just the same way as a (reentrant) daemon. It sleeps until an operation is to be performed on its data; then it wakes up, launches a thread to perform the operation, and goes back to sleep.

The concept of threads is of great value in a system that supports object-oriented programming. The most obvious interpretation is to implement a class or an object as a **daemon** (heavy-weight process) and to implement an operation on an object as a **thread** through this heavy-weight process.

6.2.2. Classes and Objects

6.2.2.1. Classes

An **object class** is an abstract data type represented as a triple:

⟨ initialization code , data type template , set of operations⟩.

In the subsequent discussion classes are denoted by Greek letters, e.g., α, β.

A class is defined either by the kernel at start-up time, or from within user programs. The definition is in the form of a number of tables, similar to *load tables* used in traditional operating systems. Once a class definition is submitted to the **Object Manager** of the system, a heavy-weight process is created; this process represents the class (it is called the **Object Coordinator** of the class). Then, one thread through the process is started: the initialization code of the class. Note that there is no restriction on the contents of the initialization code. It may contain the entire program, which will be the case if the program is a self-contained computation which does not cooperate with the outside world; it may also be empty, if the class is defined solely for the purpose of creating objects.

6.2.2.2. Objects

An instance of a class will be called **object** for short and will be denoted by a capital Roman letter in italics. Thus, $P \in \alpha$ means that P is an instance of the object class α.

An object is a data structure generated from the data type template defined as part of the definition of the class. As the declaration of an object may have run-time parameters, different objects of the same class are not necessarily identical; but they always have identical properties from the point of view of the outside world.

The definition of a class contains a set of operations that may be applied to objects of the class. These operations may be quite simple, or they may be arbitrarily complex, e.g., whole programs. For completeness, this set of operations is artificially augmented by all the operations that are performed on the class itself, such as *create subclass* or *terminate*.

An operation on an object of a class (or on the class itself) is performed by a thread. From now on, the term thread will be used to denote operations on objects or the class itself. $t(A)$ stands for an operation represented by thread t performing on entity (object) A.

6.2.2.3. Requests

In an object-oriented environment, performing a computation amounts to executing a sequence of operations on classes and objects of classes. Some systems view an operation as an atomic step, but such an approach would not be appropriate in **FLEX**, which is a system for distributed applications. In **FLEX**, several operations on objects may be in progress concurrently (or even simultaneously, if many processors are used). Moreover, one object may be the target of several concurrent operations. As mentioned above, these operations are implemented as light-weight threads.

An operation is performed on demand, i.e., there must be an entity (a thread) that wants this operation to be performed. In order to achieve this, the thread must be able to communicate this need to the daemon which executes the desired operation. It does this by sending a **message**, which contains the description of the requested operation. In **FLEX**, all the messages are in the form of operation requests. In order to avoid any misinterpretation, the term *message* will not be used; instead, the term **request** will be used.

When created by a thread, a **request** is a block of memory containing the following fields:

- a capability, which describes the operation and the target object (in encrypted form),
- optional arguments relevant to the operation.

After a request is created, it is submitted for delivery (i.e., it is sent). When it passes through the **Communication Manager**, the capability is decoded to determine the receiver (the daemon of the target object), the target object, and the name of the operation to be performed. The **Communication Manager** replaces the capability with a plain name of the operation. Then, it interrupts the target daemon, forcing it to accept the request.

6.2.2.4. Additional Considerations

Some applications require that blocks of data be passed from object to object. An example would be a matrix-manipulation package in which an object (a matrix) is assigned a new value. In **FLEX**, such an operation is considered to be a request[4] to the class daemon: a thread will be started, and it will accept the block of data and store it in the matrix. Thus, an object is

[4]In the sequel, the term **request** will have two somewhat different meanings: an **operation invocation** addressed to an object coordinator, or a **block of data** addressed to an object.

a totally passive data structure; all the processing abilities are part of the daemon, in the form of operations.

As a corollary to the above, it is not possible to send old-fashioned text messages to objects directly: instead, they must be sent to their daemons in the form of requests, as in:

$$\langle \textbf{encrypted}(\ \delta\ ,\ O\ ,\ \textbf{read}\ )\ ,\ \textit{text of message}\ \rangle$$

where O and δ denote the target object and its class daemon, respectively.

The daemon accepts this request and creates a thread, which processes the text message. This scheme allows the use of any scheme of message passing in user applications.

In some special situations, a thread may send a request to establish a direct communication line[5] with an object. Such a request is needed only if the thread does not have the capability to perform operations on the object directly and can only identify the target object by a name in plain text (as opposed to an encrypted capability).

In such a situation, it may happen that there is no object bearing the specified plaintext name. This may be due to scheduling decisions or timing. If a daemon receives a request involving an object that has not yet been created (as opposed to an object that was deleted), it does not reject such a request, but stores it, expecting that the object will be created at a later date. Garbage collection will discard such requests periodically.

6.2.3. Special Classes

As **FLEX** is an operating system to support object-oriented applications, it should treat all the entities that form it as classes and objects. There are three major parts of the system

[5]E.g., to set up a virtual circuit or some other form of handshaking.

that do not fit into the class/object concept nicely: the kernel itself, the loader (**exec** in *UNIX*) and network communication.

It is natural to try to masquerade these three entities as classes (and objects), even though they do not fit the appropriate definitions too well, as their purpose is not to serve as a vehicle for creating objects.

6.2.3.1. Class κ

The kernel is made of a number of independent modules and interrupt handlers. Internally, these modules communicate in a variety of ways, but mainly by issuing system calls (*svcs*). This method of communication is not available to user programs, as the recipient of a system call must be part of the kernel.

In order to make interprocess communication uniform in appearance, the kernel of **FLEX** is treated as a very special class κ. This class, like all the other classes, is a triple:

⟨ initialization code , data type template , set of operations ⟩

with the initialization code representing the bootstrap code, the data type template describing the hardware configuration of the system, and the set of operations representing the set of system services offered by the kernel.

Interrupts result in the creation of new threads in the kernel; this fits the conventional approach of the kernel being reentrant. There is no need to worry about the fact that the kernel is actually made of a number of independent daemons, and not of just one heavy-weight process. From any point of view, it amounts to the same, as all of them have the ability to execute in kernel mode, i.e., access the data of all the other daemons; thus, they share the same context, like threads in a heavy-weight process.

6.2.3.2. Class Π

Initially, two predefined classes exist — these classes are defined by the kernel at system initialization time. One of them is κ, the kernel itself; the other is the very important root class of the user class hierarchy. It will be denoted by Π and contains the declaration of a number of operations, such as **load** an executable file and a large variety of object-management and communication-control operations. It should be noted that Π is not a class containing privileged code; however, the execution of an operation in Π may result in kernel calls.

Class Π is an umbrella class that enables users to declare their application software. As a result of executing the **load** operation, a new object of class Π is created. It corresponds to the traditional notion of executing a user program. This new object most likely contains a number of object class definitions. Unlike other object classes, Π is not a template for identical objects; every object of class Π represents an independent user program[6].

The set of all object classes currently in existence (excluding κ) forms a tree rooted at Π. We call it **class-tree**.

When the kernel creates Π, the initialization code of Π is executed. This code contains the declaration of a number of object classes, such as a *stub class*, a *file class*, and a number of *application classes*.

One of the primary responsibilities of Π is to enable threads to communicate with each other, locally or remotely. Thus, Π contains a number of communication operations. These

---

[6]One could also claim that an object of Π is a control block similar to the PCB (Process Control Block) used in traditional systems.

operations provide access points to the **Communication Manager**.

As Π contains a library of communication operations, threads executing on objects of class Π and its sub-classes may have the right to perform these operations. This right may be inherited by their descendants.[7]

The library support can be divided into two categories:

(1) local request[8] passing, which is accomplished *via* manipulation of page table entries representing pages in local memory,

(2) remote request passing, which results in RPCs on objects residing on different machines *via* the underlying network software (*IEEE 802 standards*) and hardware (*Ethernet*).

6.2.3.3. Stub Class

Communication between two different machines (*via* a network), Π_1 and Π_2, uses a **virtual edge** connecting the two objects of their **stub classes**. For Π_1 and Π_2, the other tree is a single node represented by one object of its **stub class**. Figure 6.2 illustrates the **FLEX** system architecture.

The stub class is created during the initialization of class Π. Its responsibility is to provide transparency for network communication. The initialization code of the stub class creates a number of stub objects. Each stub object represents one network port. For different network connections, the objects may be different.

---

[7]The operations inherited may be reduced to a subset of the library.

[8]**FLEX** supports only one form of communication: passing requests to execute an operation on an object.

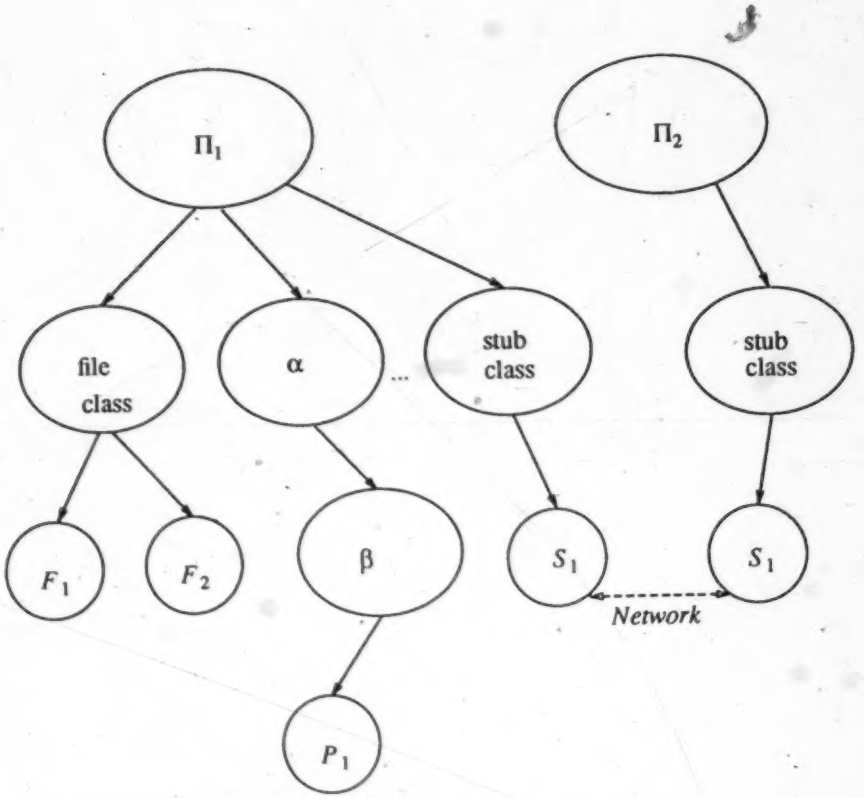

Figure 6.2: **FLEX** System Architecture

Whenever a connection between the machine and a remote site is necessary, a thread through the stub daemon is started; it operates on the corresponding stub object. The stub daemon manages the connection between the machine that it resides on and the network that the machine is connected to. The job of the stub class[9] is to transform a local request to one or more network packets and send the packets to the network. It also accepts data from the network and translates them into local requests.

[9]Or, more accurately, the job of a thread through the stub daemon.

All remote requests must go through a stub thread. Broadcast mode is used in transmitting requests to the network. When the stub daemon receives a network request, it determines whether the request is relevant to the site, or should be ignored. This may involve accessing kernel data.

The necessity to broadcast is due to the design of **FLEX**. If thread $t(A)$ sends a request to β, then β is searched for locally first. If the search fails, that means β resides on a different machine, then the request is sent through the network. Since the exact location of β is unknown, the request must be broadcast to all the machines on the system. β is then looked for in each machine. The communication between $t(A)$ and β is established after β is located.

6.2.4. Life Cycle of Classes and Objects

A class starts its existence when it is defined by some entity in the system. It is represented by a daemon (**OC**) which exists through the life duration of the class. Once a class is in existence, objects of this class may be created. Then, operations may be performed on these objects, as well as on the class as a whole. Eventually, objects of a class will be deleted, either one at a time or all at once. It is also possible to delete the class as a whole, which also deletes all its objects.

While a class and its objects live, operations may be executed on them. A thread that wants to execute such an operation must possess the right to do so; this right is in the form of a capability. Besides all the features related to object protection etc., capabilities have one other very useful property: they can be used for garbage collection: if an object is not "pointed to" by any capability, it can be deleted, as it can not be reached any more, and is useless.

The information needed to perform garbage collection is not easily available. The **Communication Manager** is responsible for it, but it has to rely on the support of **Object Coordinators** of all the classes. As the code of the **Object Coordinators** is supplied by application programmers, it may be incorrect; thus, garbage collection will either be imperfect, or will require an additional restriction: capabilities passed at run-time are revoked automatically when their original owner terminates.

Note that the design of **FLEX** provides a very sophisticated hierarchy for object manipulation. User-level applications may choose to take advantage of these functions, or ignore them. For example, a simplistic program may be implemented as the initialization code of a class with an empty set of object operations and, obviously, no objects whatsoever.

6.2.4.1. The Object Manager

The kernel module **Object Manager** is responsible for creating and deleting objects.

- The **Object Manager** creates a new object by allocating the appropriate resources to the requesting object coordinator. Note that object creation is triggered by a request issued by the object coordinator of the object to be created. This request is, in turn, triggered by a request to create the object sent to the coordinator by another entity.
- The **Object Manager** is also responsible for global clean-up after an object is deleted, while the appropriate OC is responsible for the local clean-up.

Note that in the future, the **Object Manager** will have to be augmented, so that it can perform more complex operations than those discussed.

6.2.4.2. Class Creation

When a thread executes, it may need to define a new class. The operation of defining a new class is performed in a number of steps, as shown in the following example, which assumes that thread *t*, executing an operation on an object of class α, wants to define a sub-class.

- *t* assembles together the tables containing the definition of a new class.
- *t* sends to α a request to perform the operation **create subclass** with these tables as argument.
- α processes the tables and sends a request to the **Object Manager** to define the new class.
- the **Object Manager** creates the new class — let it be called β. It then requests the **Communication Manager** to create **capabilities** to perform the operations stated in the definition of β.
- then, the **Object Manager** returns to α the capabilities.
- α returns the capabilities to *t*.

Operations on the new class β and its instances can be requested by any thread that has a copy of the capabilities created while the class was being defined.

The notation: $\beta \subset \alpha$ means that class β is a **sub-class** of class α, i.e., that some object of class α declared class β, possibly indirectly.

When a new class is defined, three new entities are created:

(1) A set of **load tables** containing the description of the code of the operations that can be performed on objects of this class, as well as on the class itself,

(2) A set of **data tables** containing the description of the data type of the template for an object,

(3) A special daemon called the **Object Coordinator** of this class. The code of OCs is not generic — a part of it is defined by the user as part of the application. In other words, users can write part of their own OC code for the classes that they want to create.

The OC daemon represents the class and all of its objects in all interaction with the outside. The life of an OC lasts precisely as long as the life of the class; in other words, the class and its OC daemon are synonymous.

6.2.4.3. The Role of the Class Daemon

The role of the class daemon (OC), depends on the attitude towards the relationship between the class as a whole and its instances. Instances of this class are called **objects**. But what exactly is part of each object and what is common to the whole class? There are two very different interpretations.

- One may treat an object as a passive data structure that is manipulated by the class daemon. From the point of view of the kernel, there is only one process per class: the demand-driven class daemon. Whenever there is a need to perform an operation on an object, the daemon is awaken – a new **thread** representing the operation is created. At any given time, many concurrent threads may pass through the daemon; a thread disappearing only when the operation is completed.

- One may also view every object as an entity that is independent from its siblings and, to the outside world, from its class daemon. The role of the class daemon is reduced to that of bookkeeping. There are as many processes as there are objects. An operation on the object is performed by the object itself. The object controls its activities by reading operation requests (in the form of messages) and processing them.

The first interpretation is more elegant than the second and should be adopted in FLEX[10].

In the first interpretation, an operation request is passed in the form of a request to create a new thread. The request is sent to the OC of the object to which the operation is to be applied. The OC is responsible for reading arriving requests and processing them in some order. This gives a simple global mechanism for executing several operations on the same object concurrently.

Note that there are other kinds of requests passed to an OC, e.g., a request for creating a new object or a block of data to be sent back as a result of an operation. These requests also result in the creation of new threads, although they do not execute on an existing object.

6.2.4.4. Object Creation

When a thread executes, it may need to create a new object of a class. This is done in a manner similar to the process of defining a class, as shown in the following example. This example assumes that thread t executing an operation on an object of class α wants to create an object of class β.

---

[10]However, as the current implementation is embedded in *UNIX*, it is very difficult to implement, as *UNIX* does not know the concept of threads.

- t builds a request frame containing the capability (identifies β and the *create object* operation), the name for the new object in the argument field followed by, if needed, additional arguments to the object creation operation.
- t sends to β the request to perform the *create object* operation.
- β processes the request frame and creates another request frame: a request to the **Object Manager** to allocate the resources needed to create the data structure that forms the object.
- the **Object Manager** allocates the resources needed to create the object.
- the **Object Manager** gives to β the resources.

6.2.4.5. Executing an Operation

When a thread wants to execute an operation on an object, it creates a request and passes it to the **Communication Manager**, which in turn passes it to the target class daemon by forcing a software interrupt upon it. The requested operation may be *trivial*, when its execution does not require any synchronization with any other thread operating concurrently (such as returning the time-of-day), or *non-trivial*, if its immediate and unconditional execution may lead to concurrency errors.

If the operation is trivial, the class daemon executes the operation immediately without consulting the kernel. On the other hand, if the operation is non-trivial, the class daemon must create a new flow of control (**thread**) that will be capable of running concurrently with other threads operating on the same object.

In the case of a non-trivial operation, the class daemon can not create a new thread itself, as the existence of a thread must be known to the kernel (a thread is a unit of scheduling). Therefore, the class daemon asks the kernel to create the new thread. It decomposes the operation request and converts it into another request: requesting the kernel to create the thread. A special kernel module, the **Thread Manager**, creates and terminates light-weight threads.

Note that in this design, a capability represents an unconditional right to start an operation on an object (i.e., a thread). In particular, OCs do not enforce access rights, as it is a capability based system.

When a request reaches its target class daemon, its address field has already been decrypted, i.e., the operation and the target object are in plaintext. The class daemon inspects this request and determines which operation is to be performed and which object (if any) is its target. If the operation is trivial, the daemon executes it immediately and goes back to sleep. If it is non-trivial, the daemon builds a new request frame containing the partial context of the thread to be created:

- a program counter pointing to the beginning of the code of the operation,
- a stack counter pointing to the top of the stack to be used by the new thread,
- an appropriate content for the remaining CPU registers, including the PSW.

When the new frame is ready, the daemon passes it directly to the **Thread Manager** by issuing an *svc*. No capability or other protection mechanism is needed here, as the daemon has an unrestricted right to create a thread through itself[11]. The **Thread Manager** creates the thread by building the new partial context. Then, it inserts the thread into the queue of threads

[11]Subject to general limitations, such as a system-wide limit on the total number of threads.

ready for execution.

6.2.4.6. The Dispatcher

Threads are the basic units of scheduling. All the threads that are ready to execute (i.e., are not waiting for an event) are grouped in a ready queue. This queue is accessed by the **Scheduler** which is a module residing outside of the kernel.

The decisions of the **Scheduler** are communicated to a kernel module called the **Dispatcher**. Based on the decisions made by the scheduler, the **Dispatcher** assigns the CPU to an appropriate thread. This is done by forcing a context switch.

6.2.5. Example

Many programs require multi-dimension arrays during their execution. Commonly, the sizes of these arrays are not known at compilation time; they are derived at execution time instead.

To create an $n \times n$ matrix, an operation $t(O)$ will start the following sequence of events:

- $t(O)$ sends a request to the OC of class α, the class defining multi-dimensional arrays. This request asks for the creation of a class, say $\alpha_2(n,n)$ of $n \times n$ arrays.
- α creates a thread that will be responsible for the creation of the requested sub-class.
- The new thread of α checks whether such a class has already been created. If not, the thread of α creates sub-class $\alpha_2(n,n)$ and its object coordinator. It also sends back to $t(O)$ the capabilities needed to manipulate this sub-class. Then, the thread terminates.
- $t(O)$ sends a request to $\alpha_2(n,n)$, asking for the creation of an instance of this class.

- $\alpha_2(n,n)$ creates an instance of its class. Let this object be called A.
- Now, if $t(O)$ wants to perform the assignment: $A[i,j]=1$, it sends to $\alpha_2(n,n)$ the request: ⟨**encrypted**(α_2 , A , **assign**) , $i, j, 1$⟩
- When $\alpha_2(n,n)$ receives this message, it creates a new thread; this thread will perform the **assign** operation.

6.3. Overview of the Communication Design

The basic communication scenario is very close to what has been described in Chapter 3. A request must be sent to an OC, whether it is asking the OC to create a new thread or it is actually a block of data sent back as a reply to a thread (operating on an object of the class that the OC stands for) in progress. In the sense, *operations* in the previous design are like *threads* in this proposed version of **FLEX**. Note that the communication primitives remain the same, and *Supermailboxes* still exist in OCs.

6.3.1. Message Objects

The previous sections presented some possible changes in the existing design. There are other ideas that could be adopted in **FLEX**, not for the purpose of changing its behavior, but for improving efficiency.

One such idea is to introduce the concept of **message objects** forming special classes. This does not introduce a new perspective to **FLEX**; it is compatible with the existing system architecture.

A message class will be denoted as μ; it must be created before the concept of a message becomes necessary for applications, e.g., at system initialization time. Each machine can have

more than one message class. Each class may specify a message type.

μ is a very special class. There are no operations defined for this class. If $t(A)$ wants to send a message to $t(B)$, a message object M of class μ must be created. Note that no messages can be sent to M, and M can not send any messages, either. It is best to view message objects as a collection of memory buffers manipulated by the kernel. Their existence is a result of taking advantage of virtual memory management.

M consists of one or more pages. The number of pages allocated for M depends on the size of the message to be sent. After it is created, M is passed from the sender to the receiver, in a number of steps. This is done by changing *page table* entries. First, the sender acquires a block of memory for M. It writes in it the desired message. Then the sender releases M to the system. M disappears from the address space of the sender (it is in transit). At some later time, the kernel puts the page table entries for M into the supermailbox of the receiver OC. Because of the concept of heavy-weight process, the threads within one heavy-weight process share the same address space; thus, shared memory may be used for getting a message to a thread once the message has arrived at the supermailbox of the corresponding class daemon. As a result, no movement of data really occurs for a local message transfer. This is significant in achieving high performance.

If a message has to be sent across the network, the actual data must be transferred through the network. Memory references used in a local environment do not work for remote message passing. The messages needed to be sent to the network stay first in the supermailbox of the stub class. From there, they are passed to remote sites.

There are two alternatives in managing M:

(1) M is created to allow both *read* and *write*. The sender produces in it a message (possibly editing it *in situ*); then it releases M to the system. The message is then passed to the receiver, which can read and modify this message at will. When the receiver does not need the message space any more, it returns it to the system.

(2) When M is first created for the sender, *write only* is allowed. Then, when it is released from the sender, it becomes *read only*. The operating system is responsible for protecting the memory access to a message object.

Simplicity of an operating system is desirable. Therefore, option (1) is chosen for **FLEX**. With this option, a more efficient message passing mechanism is also feasible, that is to use the same piece of memory for both *request* and *reply*. If $t(A)$ sends M to $t(B)$, $t(B)$ can write its reply over M and then send it back to $t(A)$. With option (2), M must be allocated each time a new message is created.

CHAPTER 7

Conclusions

The **FLEX** project is an attempt to investigate the architectural as well as algorithmic problems associated with integrating distributed operating systems and distributed database management systems. This thesis presents one of the components of **FLEX**: its focus is on computer communication in an object-oriented system. Originally, the scope of the work was restricted to implementing a communication facility for **FLEX**. While the work progressed, it became apparent that some of the decisions made initially could not be retained for reasons that became visible only at later stages of the project. Therefore, the thesis refers to *two* designs of **FLEX**: the original design with which the work started, and a modified design, which emerged as the implementation details were (slowly) being worked out.

The thesis starts by presenting some background material on network communication and distributed operating system design. Then, it presents the initial design of **FLEX**, followed by the detailed design and implementation of one of the major components of the **FLEX** kernel, the communication module.

Finally, chapter 5 contains a list of proposed improvements to the original design. These modifications are based on the experience gained during the course of the work.

7.1. Notes on the Prototype Implementation

An operating system has to run on a bare machine (or a virtual machine). At present, **FLEX** is implemented as a prototype system running as a collection of user processes under

the control of a time-sharing operating system (*UNIX*). User processes can not execute privileged instructions nor can they interface with the hardware directly. Because of this inability to reach the hardware and of the lack of privilege instruction support, the efficiency of the system is difficult to measure. The usefulness of the system can not be proven at this point.

The design of an operating system is affected by the underlying machine architecture, particularly in the areas of:

- multiprogramming (if more than one CPU is used),
- virtual memory management,
- remote communication,
- logical organization of permanent storage (commonly referred to as *file system*).

At this point it is not clear which machines **FLEX** will be running on, therefore the design and implementation adopted eventually may change drastically from what has been presented in the thesis.

For instance, if the system were to be implemented on a machine that supports efficient paged segmentation, many of its aspects should probably be replaced by the use of segments, as in Multics ([Organ-72]).

7.2. Experience Gained from the Research

This research had as its goal the implementation of a prototype object-oriented operating system. This system was expected to provide adequate support for distributed DBMS applications. A research project of this magnitude can not be done by one person working alone.

Thus, it was divided into smaller sub-projects, each of them dealing with one aspect of the whole system.

When several people work on the same project, invariably there are different ideas and points of view. Before the work on my part of the project started in earnest, a common system design was mutually agreed upon. The validity of this design was proven by the work of Christina Lau ([Lau-88]) on object management and scheduling. However, as subsequent work on communication showed, this design was not perfectly suited for communication purposes. Therefore, this thesis contains a chapter proposing modifications and enhancements of the original design.

While it is clear that the performance of communication affects directly the performance of the entire operating system, it is only one of its many components. Therefore, it is necessary to balance communication performance with the performance of other components. Moreover, the very objective of creating a high-performance system should not outweigh other desired properties, in particular simplicity, usability, and extensibility.

Work on the project exposed another important dichotomy in operating system design. There is no doubt that a reliable communication facility is a highly desired feature. It makes application programming easier since there is no need to be concerned with missing messages, etc. However, a system like **FLEX** requires another important feature: message broadcasting. As it turned out, it was not possible to implement broadcasting efficiently using reliable message passing. A less reliable, but more efficient scheme was used instead.

On a personal note, work on the **FLEX** project exposed me to many ideas in operating systems. I acquired a much stronger understanding of such concepts as: capabilities, tasks and

threads, plus a number of issues in message-based computer communication. It also gave me a better understanding of the basic concepts of object-oriented methodology.

7.3. Recommendations for Future Work on the Project

It seems that the design of **FLEX** reached a stage where the next logical step is to implement it as a stand-alone operating system, either on a bare machine or on a virtual machine.

The above is not meant to imply that the prototype is complete. But it became difficult to do further work on a prototype that runs under the control of another operating system, as many useful ideas can not be implemented. For example, it is not possible to work on memory management, network interface, file system, etc. (they are monopolized by *UNIX*). Further design work is needed in these areas, and also on object manipulation.

The existing system can be expanded by incorporating several useful features. A list of such features could include the following:

(1) Load balancing, which is a desired feature of any distributed system. Although in the current design of **FLEX** object migration is not supported, it should eventually be feasible. Note that some design features of **FLEX** are derived with it in mind, for example, the naming scheme.

(2) The support of high level languages (such as Eden), in which objects and classes can easily be expressed, should be provided to make the system practical.

Besides all of the above, there is a need for an object-oriented distributed DBMS implemented on top of **FLEX**. This would make a validation and a performance study possible.

Bibliography

[AlBLN-85]
Almes, Guy T., Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe, "The Eden System: A Technical Review", *IEEE Trans. Software Eng.*, SE-11, 1 (Jan. 1985), 43-58.

[Bach-86]
Bach, Maurice J., *The Design of the Unix Operating System*, Prentice-Hall, Inc., 1986.

[BBBCGRTY-87]
Baron, R. V., D. Black, W. Bolosky, J. Chew, D. B. Golub, R. F. Rashid, A. Tevanian, Jr., and M. W. Young, *MACH Kernel Interface Manual*, Department of Computer Science, Carnegie-Mellon University, 1987.

[Black-85]
Black, Andrew P., "Supporting Distributed Applications: Experience with Eden", *ACM SIGOPS Proceedings of the 10th Symposium on Operating Systems Principles*, 19,5 (Dec. 1985), 181-193.

[BlMaM-87]
Blair, Gordon S., Jon R. Malone, and John A. Mariani, "A Critique of UNIX", *Softw. Pract. Exper.*, 15,12 (Dec. 1987), 1125-1139.

[BrMaR-85]
Brownbridge, D. R., L. F. Marshall, and B. Randell, "The Newcastle Connection or UNIXes of the World Unite!", *Softw. Pract. Exper.*, 12,12 (Dec. 1982), 1147-1162.

[Cheri-84_1]
Cheriton, David R., "The V Kernel: A Software Base for Distributed Systems", *IEEE Software*, 1,2 (April 1984), 19-42.

[Cheri-84_2]
Cheriton, David R., "An Experiment Using Registers for Fast Message-Based Interprocess Communication", *ACM SIGOPS Operating Systems Review*, 18,4 (Oct. 1984), 12-20.

[CheZw-83]
Cheriton, David R. and Willy Zwaenepoel, "The Distributed V Kernel and Its Performance for Diskless Workstations", *ACM SIGOPS Proceedings of the 9th Symposium on Operating Systems Principles*, 17, 5 (Oct. 1983), 129-140.

[CooDr-88]
Cooper, Eric C. and Richard P. Draves, "C Threads", *Technical Report, CMU-CS-88-154*, Department of Computer Science, Carnegie Mellon University, 1988.

[FitRa-86]
Fitzgerald, Robert and Richard F. Rashid, "The Integration of Virtual Memory Management and Interprocess Communication in Accent", *ACM Trans. Comput. Syst.*, 4,2 (May 1986), 147-177.

[Goldb-83]
Goldberg, Adele, *Smalltalk-80: the Language and its Implementation*, Addison-Wesley Publishing Company, Inc., 1983.

[Gray-78]
Gray, J. N., "Notes on Data Base Operating System", *Operating Systems: An Advanced Course* Ed. Bayer, R. et al., Lecture Notes in Computer Science 60, Springer-Verlag, 1978.

[IEEE-84]
The Institute of Electrical and Electronics Engineers, Inc, *An American National Standard, IEEE Standard for Local Area Networks: Logical Link Control*, IEEE, 1984.

[Jones-78]
Jones, Anita K., "The Object Model: A Conceptual Tool for Structuring Software", *Operating Systems: An Advanced Course*, Ed. Bayer, R. et al., Lecture Notes in Computer Science 60, Springer-Verlag, 1978.

[Jones-88]
Jones, Vincent C., *MAP/TOP Networking*, McGraw-Hill, Inc., 1988.

[JonRa-86]
Jones, Michael B. and Richard F. Rashid, "Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems", *OOPSLA 1986 Conference Proceedings*, Sep. 1986, 67-77.

[Lau-88]
Lau, Christina, *Object Management, Protection and Scheduling in FLEX*, M.SC. thesis, University of Alberta, 1988.

[LCCPW-75]
Levin, R., E. Cohen, W. Corwin, F. Pollack, and W. Wulf, "Policy/Mechanism Separation in Hydra", *ACM SIGOPS Proceedings of the 5th Symposium on Operating Systems Principles*, 9,5 (Nov. 1975), 132-140.

[LLAFFV-81]
Lazowska, E. D., H. M. Levy, G. T. Almes, M. J. Fischer, R. J. Fowler, and S. C. Vestal, "The Architecture of the Eden Systems", *ACM SIGOPS Proceedings of the 8th Symposium on Operating Systems Principles*, 15,5 (Dec. 1981), 148-159.

[Marti-89]
Martin, James, *Local Area Networks*, Prentice-Hall, Inc., 1989.

[MetBo-76]
Metcalfe, Robert M. and David R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks", *Comm. ACM*, 19,7 (July 1976), 395-404.

[Meyer-87]
Meyer, Bertrand, "Reusability: The Case for Object-Oriented Design", *IEEE Software*, 4,2 (March 1987), 50-64.

[MulTa-84]
Mullender, Sape J. and Andrew S. Tanenbaum, "Protection and Resource Control in Distributed Operating System", *Computer Networks*, 8,5/6 (Nov. 1984), 421-432.

[MulTa-85]
Mullender, S. J. and A. S. Tanenbaum, "The Design of a Capability-based Distributed Operating System", *ACM SIGOPS Proceedings of the 10th Symposium on Operating Systems Principles*, 19,5 (Dec. 1985), 51-62.

[Nelso-81]
Nelson, Bruce Jay, "Remote Procedure Calls", *Technical Report, CMU-CS-81-119*, Department of Computer Science, Carnegie Mellon University, 1981.

[NiBlW-87]
Nicol, John R., Gordon S. Blair, and Jonathan Walpole, "Operating System Design: Towards a Holistic Approach?", *ACM SIGOPS Operating Systems Review* 21,1 (Jan. 1987), 11-19.

[Organ-72]
Organick, E., *The MULTICS System*, MIT Press, 1972.

[OzLLT-88]
Ozsu, M. Tamer, Christina Lau, Yan Li, and Meei-Fen Teo, "The Architecture of FLEX: A Distributed Database Operating System Testbed", *Technical Report TR88-4*, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, April 1988.

[Ozsu-88]
Ozsu, M. Tamer, "Distributed Database Operating Systems", *Technical Report TR88-2*, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, Feb. 1988.

[Pasht-82]
Pashtan, Ariel, "Object Oriented Operating Systems: An Emerging Design Methodology", *Proceedings of the ACM'82 Conference* (Oct. 25-27, 1982), 126-131.

[PopWa-85]
Popek, Gerald J. and Bruce J. Walker, *The LOCUS Distributed System Architecture*, MIT Press, 1985.

[Rashi-86]
Rashid, R. F., "Threads of a New System", *Unix Review*, 4,8(Aug.1986),37-49.

[RasRo-81]
Rashid, Richard F. and Geroge G. Robertson, "Accent: A Communication Oriented Network Operating System Kernel", *ACM SIGOPS Proceedings of the 8th Symposium on Operating Systems Principles*, 15,5 (Dec. 1981), 64-75.

[Sanso-88]
Sansom, Robert Daniell, "Building a Secure Distributed Computer System", *Technical Report, CMU-CS-88-141*, Department of Computer Science, Carnegie Mellon University, 1988.

[SilPe-88]
Silberschatz, A., and J. Peterson, *Operating System Concepts*, Addison-Wesley Publishing Company, Inc., 1988.

[Stall-87]
Stallings, William, *Local Networks*, Macmillan Publishing Company, 1987.

[Tanen-81]
Tanenbaum, Andrew S., *Computer Networks*, Prentice-Hall, Inc., 1981.

[TanvR-85]
Tanenbaum, Andrew S., and Robert van Renesse, "Distributed Operating Systems", *ACM Computing Surveys*, 17,4 (Dec. 1985), 419-470.

[TanMu-81]
Tanenbaum, Andrew S. and Sape J. Mullender, "An Overview of the Amoeba Distributed Operating System", *ACM SIGOPS Operating Systems Review* 15,3 (July 1981), 51-64.

[WCCJLPP-74]
Wulf, W., E. Cohen, W. Corwin, A. Jones, R. Levin, C. Pierson, and F. Pollack, "HYDRA: The Kernel of a Multiprocessor Operating System", *CACM*, 17,6 (June 1974), 337-345.

[WPEKT-83]
Walker, B., G. Popek, R. English, C. Kline, and G. Thiel, "The LOCUS Distributed Operating System", *ACM SIGOPS Proceedings of the 9th Symposium on Operating Systems Principles*, 17,5 (Oct. 1983), 49-70.

[WuLeH-81]
Wulf, W. A., R. Levin, and S. Harbison, *Hydra/C.mmp: An Experimental Computer System*, McGraw-Hill, 1981.

Material used, but not referenced

[ArChF-87]
Artsy, Yeshayahu, Hung-Yang Chang, and Raphael Finkel, "Interprocess Communication in Charlotte", *IEEE Software*, 4, 1 (Jan. 1987), 43-28.

[BarLi-85]
Barak, Amnon and Ami Litman, "MOS: A Multicomputer Distributed Operating System", *Softw. Pract. Exper.*, 15,8 (Aug. 1985), 725-737.

[BirNe-84]
Birrell, A. D. and B. J. Nelson, "Implementing Remote Procedure Calls", *ACM Trans. Comput. Syst.*, 2,1 (Feb. 1984), 36-59.

[BroWu-86]
Brown, Geoffrey and Chuan-lin Wu, "Operating System Kernel for a Reconfigurable Multiprocessor System", *IEEE Proc. 1986 International Conference on Parallel Processing*, 234-241.

[ColLe-82]
Collins, John P. and Miles M. Lewitt, "Object Oriented Operating System for Microcomputers", *Computer Design*, 12,6 (June 1982), 165-172.

[Kaare-83]
Christian, Kaare, *The UNIX Operating System*, John Wiley & Sons, Inc., 1983.

[LeGeC-84]
LeBanc, Thomas J., Robert H. Gerber and Robert P. Cook, "StarMod Distributed Programming Kernel", *Softw. Pract. Exper.*, 14,12 (Dec. 1984), 1123-1139.

[Lisko-82]
Liskov, B., "On Linguistic Support for Distributed Programs", *IEEE Trans. Software Eng.*, SE-8 (May 1982), 203-210.

[Marin-86]
Marinescu, Dan C., "Inter-process Communication in MVS/XA and Applications for Scientific and Engineering Information Processing", *Softw. Pract. Exper.*, 16,5 (May 1986), 1123-1139.

[QuSiP-85]
Quarterman, John S., Abraham Silberschatz, and James L. Peterson, "4.2BSD and 4.3BSD as Examples of the UNIX System", *ACM Computing Surveys*, 17,4 (Dec. 1985), 379-418.

[Rashi-80]
Rashid, Richard F., "An Inter-Process Communication Facility for Unix", *Technical Report, CMU-CS-80-124*, Department of Computer Science, Carnegie Mellon University, 1980.

[Spect-82]
Spector, A. Z., "Performing Remote Operations Efficiently on a Local Computer Network", *CACM*, 25,4 (April 1982), 246-260.

[Strou-88]
Stroustrup, Bjarne, "What is Object-oriented Programming?", *IEEE Software*, 5,3 (May 1988), 10-20.

[Tanen-87]
Tanenbaum, Andrew S., *Operating Systems: Design and Implementation*, Prentice-Hall, Inc., 1987.